Automated mass maintenance of software assets

Veerman, N.P.

2007

Automated mass maintenance of software assets

Niels Veerman

Automated mass maintenance of software assets

Niels Veerman

VRIJE UNIVERSITEIT

Automated mass maintenance of software assets

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 15 januari 2007 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Niels Veerman

geboren te Amsterdam

promotor: prof.dr. C. Verhoef
copromotor: dr. A.S. Klusener

Dit onderzoek werd ondersteund door het Ministerie van Economische Zaken via:
This research has been sponsored by the Dutch Ministry of Economic Affairs via:

SENTER-TSIT3018 *CALCE: Computer-Aided Life Cycle Enabling of Software Assets*

aan Cindy

Dankwoord

Dit onderzoek zou niet mogelijk zijn geweest zonder de support van vele mensen gedurende de afgelopen vier jaar. Chris Verhoef en Steven Klusener hebben mij de mogelijkheid, steun en vrijheid gegeven om onderzoek te doen, en om deel te nemen aan diverse evenementen en activiteiten. Ze hebben mij op vele manieren ondersteund, variërend van technisch-inhoudelijke zaken bij het werken met grammatica's tot de redactionele ondersteuning bij het overtuigen van reviewers. Chris en Steven hebben er alles aan gedaan om voor een plezierige en inspirerende werkomgeving te zorgen, en ik wil ze graag danken voor hun continue inspanningen om me te helpen en iets te leren.

Er zijn vele andere mensen, met name binnen onze onderzoeksgroep, die op een positieve manier bijgedragen hebben aan mijn werk en werkomgeving. Een aantal van hen wil ik hier bedanken. Voor meer dan drie jaar was Ernst-Jan Verhoeven een plezierige kamergenoot. Ernst heeft niet alleen belangrijke bijdragen aan het onderzoek binnen onze groep geleverd, maar zeker ook aan de goede werksfeer, en hij voorzag mij van mijn dagelijkse dosis nuttige en minder nuttige informatie. Michel Oey heeft eveneens voor veel gezelligheid gezorgd door regelmatig even te komen kijken hoe het met mij ging, en daarbij toonde hij ook altijd interesse in mijn werk. Michel heeft me vaak geholpen met de meest uiteenlopende zaken, variërend van programmeerwerk tot opmaak tips voor artikelen. Rob Peters heeft in grote mate bijgedragen aan de goede sfeer op onze werkkamer. Rob's vele verhalen en anekdotes waren een welkome afwisseling tijdens de dagelijkse bezigheden. Verder zorgde Rob er ook voor dat er niet te lang achter elkaar gewerkt werd, door op gezette tijden een koffiepauze in te lassen. Ralf Lämmel en Mark van den Brand, voor korte tijd mijn kamergenoten, wil ik danken voor hun inbreng, steun en gezelschap. Ralf heeft altijd enthousiasme en interesse voor mijn onderzoek getoond en was altijd in voor grapjes. Mark heeft mij tijdens mijn eerste conferentie geholpen en stond altijd klaar om vragen te beantwoorden en problemen op te lossen. Met Gerbrand Stap en met Philip Pirovano, beide afstudeerders op onze werkkamer, heb ik ook een hele leuke tijd doorgebracht. René Krikhaar heeft veel tijd en moeite gestoken in het realiseren van een Postdoc plaats, en heeft voor een gezellige en vruchtbare samenwerking op onderwijsgebied gezorgd. Elly Lammers wil ik graag bedanken voor haar hulp met de vele administratieve dingen die werk met zich meebrengt. Ruud Wiggers en Gert Huisman hebben me altijd goed geholpen met uitstekende service en support op IT gebied. Hans van Vliet dank ik voor de mogelijkheid om onderwijs te verzorgen.

Verder wil ik in het kader van het CALCE-project graag de betrokken mensen van het Centrum voor Wiskunde en Informatica, van de Software Improvement Group, en van Getronics PinkRoccade hartelijk danken voor de samenwerking. De mensen die me bij een specifiek onderwerp geholpen hebben, heb ik bij het betreffende hoofdstuk vermeld.

Tot slot wil ik mijn leescommissie danken voor hun inspanningen om dit proefschrift te lezen en beoordelen, en voor hun suggesties voor verbeteringen:
prof.dr. M.G.J. van den Brand, dr. L. O'Brien, prof.dr. J. Ebert, prof.dr. W. Fokkink, en prof.dr. P. Klint.

Niels Veerman
September 2006

Contents

Chapter 1

Introduction

1.1 Software assets

Software plays an important role in society. Computers and software have evolved over the years, from early adding machines to data processing and powerful ways for communication and sharing of information. Software systems are deeply ingrained in our world, and have changed the daily lives of many. Some even state that our civilisation runs on software. Consider the software in our cell phones, televisions, and cars, and the software that enables world wide communication. Observe that many companies' existence depends on software and its proper operation. Organisations in all industries, whether is in the financial, automotive, or medical industry, cannot operate without their software. Therefore, software systems are often called *software assets*. Software assets should be treasured and require proper care, that is, appropriate maintenance.

1.2 Software maintenance and evolution

To maintain something is defined as to keep it in an existing state, or to preserve it from failure or decline by providing it with necessities for life or existence [149, 160]. For software this means that it must be adapted to the changing world in which it is being used. Since a great deal of software reflects aspects of society that are subject to change (e.g. political decisions, technological advances, business or legal requirements, economical and social developments), software must be changed as well. Furthermore, to maintain user satisfaction, inappropriate behaviour of software (e.g. exposed faults) must be repaired. The adaptive nature of changing software is often called *software evolution* [19, 138]: software is gradually developed in order to keep up with its changing environment. In fact, about 50 to 80% of the total software life-cycle expenses are consumed to maintain the evolution of the software [26, 106, 147], and thus the majority of the software engineers are modifying existing software. Hence, *changeability* is perhaps one of the most desirable attributes of software [137].

There are several categories for software maintenance activities and several classifications with different definitions [102, 189, 190]. An accepted classification is the following one:

- **Corrective maintenance:** modification of software performed to correct discovered faults, i.e. faults in requirements, design, or implementation.
- **Adaptive maintenance:** modification of software to keep it usable in a changed or changing setting. For instance, changes in hardware, compilers, operating systems, or other software that is used. This does not include new functionality.
- **Perfective maintenance**: modification of a software to add functionality to meet new user requirements, or to improve performance or maintainability.

In this classification, the implementation of new functionality to meet user requirements is classified as perfective maintenance (i.e. to come close to a perfect product), but it can also be seen as an adaptation to the changing world in which the software operates. Therefore, adaptive and perfective maintenance are sometimes joined and called *enhancements* [158, 163] or *continued development* [142].

An early study on the distribution of maintenance activities among the categories showed that about 20% is corrective, 20-25% is adaptive, and 55-60% is perfective [141, 142]. More recent studies, discussed in [163], confirm that data. However, a single maintenance activity can imply changes in several of the presented categories. For example, a user requests new functionality for a software product (perfective maintenance). This new functionality requires a new version of the used database system. The new version of the database imposes certain constraints on the software product, such that the product must be modified (adaptive maintenance). In addition, the use of the new database version reveals an error in the product, which must be repaired (corrective maintenance). So, although the initial change concerns perfective maintenance, adaptive and corrective maintenance has to be performed to actually carry out the change.

The evolution of software, resulting from maintenance changes, has led to the formulation of laws of software evolution [18, 135, 136]. The first two laws, the law of continuing change and the law of increasing complexity, state that (1) software must be continually adapted to remain satisfactory, and (2) the complexity of evolving software increases unless effort is made to maintain or reduce it. The first law reflects the dependency between software and the changing world in which it used, while the second law indicates that evolving software becomes increasingly difficult to maintain unless the growth in complexity is constrained. This is also called the software evolution paradox in [68]: evolution impedes further evolution.

The increasing complexity of evolving software is amplified by the way changes are implemented. Since the ultimate purpose of changes to software is economic gain [135], it is usually required to implement a change at minimum cost in limited time. This results in successive changes throughout the software that are accumulated during its lifetime, disregarding the original, current and coming structure of the software. This process yields an increasingly complex structure, which complicates comprehension and future changes. Unless effort is put into reducing the complexity of the structure, evolution of software causes its structure to deteriorate. However, the complexity of software is only

reflected in the time to apply changes on the long-term and not directly in the current functionality of the software. Therefore, reducing the complexity of deployed software usually has a low priority in the software engineering process.

1.3 Research on software maintenance and evolution

The insights in the evolution of software and the accompanying maintenance problems are not new and have been studied for several decades now. It has led to the existence of several academic conferences (e.g. [20, 143]), workshops (e.g. [67, 205]), and a dedicated journal ([56]) on software maintenance and evolution. Topics range from the low-level analysis of source code, refinement of requirements specification techniques, to the improvement of the maintenance process of an organisation. Software can be examined at any level of abstraction (e.g. documentation, requirements, source code, object code).

One of the research areas is *program comprehension*: the development and deployment of tools and techniques to improve the understanding and ease the modification of existing software artefacts. Program comprehension is considered to be the key to effective maintenance [156]. It is estimated that maintainers spend 40 to 60% of their time on analysing code [163], and therefore program comprehension has received considerable attention [146]. The notions software *reverse engineering*, software *reengineering*, and software *restructuring* [57, 190] were introduced:

- **Software reverse engineering:** the process of analysing software to create representations of the software in another form or at a higher level of abstraction, as opposed to forward engineering. The purpose of reverse engineering is to improve comprehension of the software by extracting certain information (e.g. an overview of the relationships between components), and can be done at any level of abstraction or stage of the life-cycle. For example, the creation of flow graphs from source code, or the recovery of the design of a software system from available artefacts.

- **Software reengineering:** the examination and alteration of software to reform it. Usually, reengineering is a combination of reverse engineering and forward engineering. First, software is reverse engineered to obtain a particular representation, and second, the obtained representation is used to perform a forward engineering step. The purpose of reengineering can be to obtain improvements in the design or implementation of a software system, or to implement new requirements.

- **Software restructuring:** changing the structure of software. The software remains at the same abstraction level, and the external behaviour of the original representation and the restructured representation are equivalent. For example, the reorganisation of a data model, the replacement of complex control structures by simpler alternatives, or vice versa to obfuscate the flow of control.

These notions are strongly interrelated and focus on techniques for analysis and manipulation of existing software. The research and practice on these topics continued, and evolving software became more and more complex. Many software systems were considered as monolithic entities that resist change: *legacy systems*. At the same time, such

software is a precious asset to business and government, and hence to society. Controlling the complexity of software became a critical success factor in software maintenance [182]. Then, with the Year 2000 problem surfacing, *software renovation* [57, 70] attracted more and more attention in order to fight the increasing complexity of evolving software. Software renovation aims at improving software to make it more comprehensible, extensible, robust and reusable. To renovate software, software reengineering and restructuring techniques are deployed. Later on, with the emerging integration of existing software into new technology, it became more and more apparent that tool-supported software reengineering is a prerequisite for extraction, wrapping, integration, redevelopment, and replacement of evolving software assets [180, 181, 183].

1.4 Massive software maintenance

At the end of the 20th century, a new class of software project emerged: mass update projects [108]. These *massive software maintenance* projects concern widespread changes to large deployed software systems, which must be carried out simultaneously. Thousands of changes must be made to millions of lines of source code. The most prominent examples are the Year 2000 changes and the Euro conversion, but there are many more. In fact, similar projects have been carried out since large-scale software systems were built; for instance, to migrate to new hardware, operating systems, databases, or to adapt to pervasive requirements. Virtually any software system that is nowadays in use and that was initiated more than a decade ago has been subject to one or more of such large-scale changes.

Massive changes have nearly always been carried out in a traditional way: *by hand*. Although it is quite natural to perform local, onetime changes by hand, the time to perform changes does not scale linearly with the amount of changes to be made and the size of an application. Pervasive changes to a multi-million line application scales beyond the scope of the day-to-day routine of normal maintenance. In particular, modifications that affect parts of an application's architecture can be pervasive, such as changes to interfaces, platforms, implementation languages and database schemas. But even a seemingly simple expansion of a datastructure, such as the Year 2000 problem, can cause widespread changes to a software application. The large impact is caused by related datastructures and functions that have to be modified as well. Another example is a change to a component that represents common functionality, such as logging, accessing a common database, error handling, or any other functionality that is used by several components. If, for instance, a parameter or the number of parameters of an interface function must be changed, each reference to that function must be updated accordingly. The modification of a common component can require many changes in various other components, resulting in numerous changes. The impact of such an initially simple looking change can easily be underestimated in practice, because interfacing software components are often interconnected in several ways. These dependencies are usually not very transparent. To modify a single component in isolation is then difficult or simply not possible, and a simultaneous update to all involved components is necessary. Nevertheless, massive software maintenance is often approached in a naive way. As the impact of massive changes are not properly

calculated, manual approaches are deployed. When the impact becomes clearer, endless patches are applied to the software to get it right.

The underestimation of large-scale changes is one of the reasons for the increasing complexity of software: changes increase the complexity of software, but manual changes complicate software even more. Humans are not good at consistently applying changes by hand over and over again to thousands or millions of lines of code, and errors and inconsistencies are introduced in the software. To make things worse, the nature of the projects and the increasingly demanding environment of the software force programmers to carry out such changes almost instantly, resulting in uncontrolled ad hoc solutions. Hence, the massive changes to evolving software that are applied by hand cause further deterioration of code.

1.5 Software maintenance automation

One way to provide support for engineering software maintenance is the deployment of tools and services to assist the maintainer, that is, to perform certain tasks automatically. To assist a maintainer in reverse and forward engineering steps that must be carried out during maintenance, tools provide support. This varies from text editors, debuggers, test-case generators, visualisation and restructuring tools, to version control and configuration management tools. Recall that maintainers spend 40 to 60% of their time on understanding code, so a tool or technique that can decrease this time is valuable. For example, because programs are often first understood in terms of procedures and control-flow [161], the use of a generated control-flow graph of the program can speed up the comprehension process. As large-scale modifications have a high degree of regularity, tools can provide support to make the changes. A single modification must usually be repeated in many places throughout an application. Therefore, a tool can be used to support or carry out massive changes in a fast, cost-effective, reliable and consistent way.

Tools to support software maintenance have been developed for decades now, but the adoption of large-scale maintenance automation in industry is still marginal [61]. There can be various reasons for this, ranging from the usability, compatibility, and flexibility of tools, to the psychological, economical, and organisational realities of adoption. Mass change projects, like the Year 2000 change, Euro conversion and many to come, have emphasised the need for automation to perform massive software maintenance in a well-engineered way.

To carry out massive maintenance, automatic software transformations can be deployed. The size and nature of a change determine the degree of automation that should be applied. An impact analysis can provide insights into the scope and frequency of a change. High frequency changes in large code volumes require a high degree of automation, but it can be unattractive to automate low frequency changes. Automatic software transformation tools are usually built by specialised parties, but tools are not meant to replace the maintainers of a software system. On the contrary, maintainers play an important role in software maintenance automation. They are domain experts having valuable knowledge of the software, which is indispensable for constructing and validating maintenance tools. In case of an automatic modification, they are the users of the end product

of the automation effort: the modified source code.

Software transformation tools usually have upfront costs, but provide benefits such as flexibility, consistency, and repeatability. Using a language's grammar, generic language technology can be used to generate basic functionalities for software transformations, such as parsers and generic transformers [44].

1.6 The message of this thesis

The core message to take away from this thesis is:

Automation of massive software maintenance helps to combat the increasing complexity of evolving software

Large-scale analyses can provide insights into software to support decision-making regarding modifications, such as the impact of a change, the measurement of complexity and detection of errors. If a problem is better understood, it can be dealt with more adequately. Analysis tools can also support the use of coding standards by detecting violations and increasing consistency throughout the software. Hence, automated analyses can be used to avoid ad hoc short-term solutions and allow more control over the long-term growth of software complexity.

Modification tools can assist in making reliable changes throughout software, and have significant advantages over traditional manual approaches: precision, control, consistency, execution time, and so on. In particular, structural modifications to software, such as mass maintenance changes, benefit heavily from automation. Automated modifications are usually cost-effective because manual modifications introduce inconsistencies and concealed errors, which deteriorate the software and cost more in the long run. Furthermore, the reengineering of evolved legacy applications, which is a prerequisite for proper evolution, can only be carried out in a reliably way using an automated approach.

1.7 Outline and contributions of the thesis

In this thesis, we pursue the capabilities of mass maintenance automation to support software evolution, that is, to allow for more control over the increasing complexity of evolving software. One of the most difficult parts of maintaining evolved source code is probably to find out *what* must be changed. If many changes must be carried out simultaneously in many interdependent programs, another difficulty is *how* to apply the changes. Automatic tools have their limitations, but they can aid to support these challenges: to find out what must be changed and the way a change is made. We employed mass maintenance tools for different purposes to investigate their value for software evolution. Furthermore, by tackling real-life problems, we also want to stimulate awareness of software maintenance automation in industry.

We start with a brief overview of some of the tools and technology suitable for massive software maintenance transformations. Then, we investigate defects and error-prone code as a result of software evolution, and we explain how automatic analysis and restructuring

can bring relief for complex code. After that, we move on to massive changes. Large-scale structural changes can cause significant increases in the complexity of software, and an automated approach provides more control over these unwanted side effects of software evolution. Subsequently, we continue with massive code restructuring to change-enable evolved software. Software restructuring is a prerequisite for proper evolution of software assets, such as extraction, wrapping, integration, redevelopment, and replacement of software. Finally, we pursue techniques to control a mass maintenance automation process itself.

Scope of the research

Estimates of the total number of lines of source code in the world range in the hundreds of billions, increasing every year. A large amount of this code has been written in the Cobol language, which has existed for over 40 years now. It was estimated in 1997 [8] that there was more than 180 billion lines of Cobol code in active use, growing at a rate of 5 billion lines a year. Some more recent data is presented in [7]:

- 75% of all production transactions on mainframes is done using Cobol
- Over 60% of all Web-access data resides on a mainframe
- Cobol mainframes process more than 83% of all transactions worldwide
- Over 95% of finance-insurance data is processed with Cobol

The huge amount of evolved Cobol code requires a great deal of maintenance; hence, Cobol is a proper subject for industrial research on mass maintenance. Although a typical Cobol application involves several languages (e.g. control languages, database definitions, screens, and so on), the scope of the research is limited to the Cobol source code of applications.

Outline and contributions

To study software evolution and software maintenance automation, we employed automated tools and techniques in industrial settings. We start with a brief overview of transformation tools and technology for mass maintenance in Chapter 2. In chapters 3, 4 and 5, we deploy tools for mass maintenance. In Chapter 6, we discuss techniques to control mass maintenance itself. Chapter 7 concludes the thesis. We summarise each chapter and its contributions here.

Chapter 2 We give a brief overview of some of the available transformation technologies and (research) tools that are suitable for mass maintenance. We touch upon lexical and syntactic approaches, generic language technology, grammar engineering, and automated maintenance factories.

Chapter 3 In the Minefield project, we perform large-scale analyses for maintenance purposes. We detect possible defects in programs by defining coding standards and analysing violations (called "minefields"), which are the result of manual evolution. By evaluating violations with a system expert, we distinguish real errors from error-prone programming styles. The detected errors were caused by inconsistencies that had been introduced by hand. Furthermore, we argue that restructuring techniques can aid in fighting complex code. High-level representations of programs illustrate the value of automation by depicting the flow of control in a program and visualising the coding standard violations.

Contributions: Definitions and analyses of several error-prone programming styles in Cobol, with a case study on five industrial systems comprising over 800 thousand lines of code. We establish a relation between violations of coding standards and possible programming errors, and we explain how complex code in evolved software can be dealt with by using restructuring techniques.

Chapter 4 We move on to the entire mass change process. In the context of the Btrieve project, an industrial mass maintenance project, we develop mass analysis and change tools, and a mass maintenance infrastructure. An entire software portfolio must be adapted to a changing environment; hence, new requirements must be implemented that require a structural change to tens of interdependent systems comprising thousands of programs. The automated approach provides a low-cost and low-risk way for this kind of evolution.

Contributions: An elaborate experience report on a commercial industrial mass update project, involving a software portfolio of 45 Cobol systems with over 4 million lines of code in total. We illustrate the requirements and advantages of an automated approach over a manual approach for a structural change to an entire software portfolio.

Chapter 5 We continue with code restructuring. In the Restructuring project, we extend and deploy a restructuring technique to improve the modifiability of legacy programs and to allow proper evolution. We argue that the approach can transform an evolved monolithic program into a more flexible program, which can be modified and evolve more easily. Loosely coupled components can be added, extracted, wrapped and integrated more easily. Two case studies with a significant code base illustrate the performance.

Contributions: Improvements and extensions to an existing Cobol restructuring algorithm, and a large-scale application in two case studies covering over 5 million lines of code. By adapting and extending the existing approach, we show that it is flexible and scalable. The case studies illustrate the wide applicability of the restructuring algorithm.

Chapter 6 We present and evaluate an approach for checking automated mass maintenance transformations. We elaborate on the difficulties when it comes to controlling automated mass maintenance changes. To detect errors in mass change tools, we propose a number of lightweight cost-effective techniques, including code analyses, bisimulation equivalence, and grammar-based testcase generation. We apply and evaluate the techniques on the mass maintenance efforts from Chapter 4 and Chapter 5.

Contributions: A lightweight approach for checking mass maintenance transformations. We present and evaluate a number of techniques to have cost-effective control over large-scale changes.

Chapter 7 We summarise the thesis and present some concluding remarks.

Origin of the chapters

Most of the material in this thesis has been published:

- **Chapter 3**
  N. Veerman and E. Verhoeven. Cobol minefield detection.
  Software: Practice & Experience, 36(14):1605-1642, 2006.

- **Chapter 4**
  N. Veerman. Automated mass maintenance of a software portfolio.
  Science of Computer Programming, 62(3):287–317, 2006.

- **Chapter 5**
  N. Veerman. Revitalizing modifiability of legacy assets.
  Journal of Software Maintenance and Evolution: Research and Practice, Special issue on CSMR'03, 16(4–5):219–254, 2004.

  N. Veerman. Revitalizing modifiability of legacy assets.
  In M. van den Brand, G. Canfora, and T. Gyimóthy, editors,
  Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03), pages 19–29. IEEE Computer Society Press, 2003.

- **Chapter 6**
  N. Veerman. Towards lightweight checks for mass maintenance transformations.
  Science of Computer Programming, 57(2):129–163, 2005.

- **Samenvatting**
  S. Klusener en N. Veerman. Automatiseren complex maar lucratief.
  Informatie, September 2003.

- **Additional publications** (not included in this thesis)
  M. van den Brand, T. Kooiker, N. Veerman, and J. Vinju.
  An architecture for context-sensitive formatting (extended abstract).
  In H.M. Sneed, T. Gyimóthy, and V. Rajlich, editors,
  Proceedings of the 21st International Conference on Software Maintenance (ICSM'05), pages 631–634. IEEE Computer Society Press, 2005.

  M. van den Brand, T. Kooiker, N. Veerman, and J. Vinju.
  A language independent framework for context-sensitive formatting.
  In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors,
  Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06), pages 103–112. IEEE Computer Society Press, 2006.

The original articles have been edited, but each chapter is self-contained and can be read independently. Hence, there can be some textual overlap between chapters.

Chapter 2

Tools and technology for massive software maintenance automation

In this chapter, we discuss tools and technology for automating massive software maintenance. When we mention software, we refer to software at the source code level. Basic tools for examining and manipulating source code include text editors, file comparators, debuggers, and so on. The purpose of this chapter is to elaborate on possibilities when it comes to choosing tools for automating a mass maintenance task as much as possible. The main ingredients for successful massive software maintenance are presented. We touch upon lexical and syntactic tools, generic language technology, grammar engineering, and the architecture that combines all these together: a software maintenance factory.

2.1 Lexical and syntactic tools

Changes to or analyses of source text can be done on several different levels. The lowest level is the lexical level, comprising characters and words. The syntactic level is the level of sentences and a language's grammar. Then there is the semantic level, which assigns meaning to a language. The hierarchy of the analysis levels is illustrated in compiler construction. The parsing phase consists of lexical and syntactic analysis [5]. A lexer scans the input and feeds the parser with tokens, consisting of characters. The parser then tries to match the tokens according to a language's grammar. Subsequently, semantic analysis can be used to verify that the input makes sense according to the language's semantics. Lexers and parsers can be generated from a language description; the most well-known scanner generator is probably Lex [139] (**Lex**ical Analyzer Generator) and the most well-known parser generator is probably Yacc [104] (**Y**et **a**nother **c**ompiler **c**ompiler). Combined, Lex & Yacc [50] can be used to build software analysis and manipulation tools, such as compilers, but also mass maintenance tools.

In software maintenance, code analysis and modification tools are usually classified

as either lexical or syntactic, but the precise border between lexical and syntactic tools is not very clear. Such tools are often used in combination with each other. A lexical tool typically uses pattern matching and replacing with regular expressions on code fragments, whereas a syntactic tool has more knowledge of a language's syntax and can replace complex code structures. Usually, a syntactic tool first parses all the code before doing an analysis or making a change; this means that context information can be used with little effort. In general, a syntactic tool is far more powerful and versatile than a lexical tool, but the choice between the two depends on several things: the task that has to be performed, the required accuracy and performance, the available technology and infrastructure, and expertise of the practitioner who has to carry out the task.

For a command-line quick but inaccurate code scan, a lexical tool like grep (**g**et **r**egular **e**xpression and **p**rint) and similar tools, such as Perl (**P**ractical **e**xtraction and **r**eport **l**anguage) [170] and sed (**s**tream **ed**itor) & awk (**A**ho, **W**einberger and **K**ernighan) [4, 74, 148], are very useful but should be deployed with care. Lexical tools have severe limitations and can easily result in false positives and false negatives, and regular expression patterns can become intricate and difficult to comprehend and maintain. We illustrate this with an simple example.

Assume we just want to count the number of goto statements in a Cobol program using `grep`. A goto statement can be recognised by the (reserved) keyword `GO`. One could implement the following simple query:

```
$ grep -c " GO " program.cob
```

With this query, all lines with the substring " `GO` " are retrieved from `program.cob` and counted. The query can be understood with little effort, and usually the result provides a proper approximation of the number of goto statements in the program. However, the `GO` keyword does not always have to be preceded or followed by a space; in some cases this is a tab, or another control character. The current query misses these cases, but an improved query with a regular expression takes them into account:

```
$ grep -c -E "[\ [:cntrl:]]GO[\ [:cntrl:]]" program.cob
```

But this new query misses goto statements that appear at the beginning of a line. Although it is not common in Cobol to start a statement at the beginning of a line, it is allowed by many compilers. To take these cases into account, the query is modified such that there is either the start of a line (`^`) or one of the other characters directly preceding the `GO` keyword:

```
$ grep -c -E "(^|[\ [:cntrl:]])GO[\ [:cntrl:]]" program.cob
```

In addition, a goto keyword can appear in mixed case (this can also be achieved by using the `-v` option of grep):

```
$ grep -c -E "(^|[\ [:cntrl:]])[Gg][Oo][\ [:cntrl:]]" program.cob
```

At first sight, it may take one some time to see what this pattern matches, but it seems to be a robust one which can handle all possible cases. Still, this is not true. Retrieved

goto statements may appear in comment lines, which are usually indicated by an asterisk or slash in certain columns, so these have to be filtered out. Moreover, several statements can appear on a single line, which are counted as one, or a statement may be spread over several lines. To accurately count these cases, a single `grep` command is not sufficient and a more complex lexical tool is required to carry out a precise analysis. Eventually, using a complex regular expression pattern, we will be able to count all the goto statements in an arbitrary Cobol program. But what if we need a more detailed analysis of the goto statements in a program. For instance, to extract certain functionality or components of an existing program, it can be necessary to determine which goto statements jump backward or forward in a program (to detect loops or dependencies). Or, if we need to perform an accurate control-flow analysis to detect coding standard violations (see the Minefield project in Chapter 3) or to calculate complexity metrics. Hence, for quick code scan, a lexical approach is very suitable, but an accurate analysis using a lexical tool becomes truly intricate, and more advanced techniques are required.

A code modification using lexical tools should be performed with even greater care, because it can result in severe problems. Where a false negative or positive in a code analysis results in an imprecise analysis, a lexical tool for a code modification can create serious problems in a software system. An elaborate example of this is discussed in [44, p 248], where the removal of a single keyword creates havoc in a production system. In that example, the seemingly simple change to the code actually required preprocessing, pretty printing and grammar knowledge. However, people tend to think in the short-term and implement a lexical tool to do a quick code manipulation. It does not seem to pay off to develop a grammar that can parse an entire program while the analysis involves only a few statements. But sooner or later, the tool must be enhanced to support more advanced transformations. The tool becomes more and more complex to support all kinds of exceptions.

A grammar-based, syntactic tool does not suffer from the above limitations, and allows for concise patterns for matching and replacing. If, for example, we want to count the goto statements in a program, we end up with simple patterns. This is because the code is parsed first, and, after it is parsed, goto statements are already classified as goto statements in the parse tree (assuming the code is parsed correctly). Then, by traversing the tree, goto statements can easily be visited and counted, and more advanced analyses can also be performed with little effort. The context details, that complicated the `grep` example, are unimportant because a syntactic approach can abstract from the surrounding characters, such as the tab or newline. Hence, these characters do not appear in a pattern, and matching multi-line code patterns can be implemented concisely (i.e. match code patterns that are spread over several lines in the program to be transformed). A transformation pattern for counting goto statements, using rewrite rules for transformations, can look as simple as this:

```
count-gotos( Goto-statement, Integer ) = Integer + 1
```

This transformation searches for goto statements in a program, and an integer is increased for each such statement. When the entire program has been traversed, the accumulated value is returned. Dedicated technology can implement the traversal mechanism

automatically. We describe such technology and the above example in more detail in the next section.

A pattern to count goto statements that jump backward can also be implemented concisely, as is shown below. For now, a Cobol program consists of sequences of labelled statements, which are called paragraphs. The pattern recursively visits a list of paragraphs, and counts all goto statements in the list referring to the label of the first paragraph. An additional function, `count-goto-lab()`, counts all goto statements that refer to a specified label. Local goto statements (i.e. loops within a single paragraph) are not counted as gotos that jump backward.

```
count-backward-gotos( Label. Statements. Paragraphs )
=
  count-goto-lab( Paragraphs, Label )
+ count-backward-gotos( Paragraphs )
```

These patterns are accurate and more intuitive than the cryptic expressions we showed earlier, because they are implemented using constructs of the programming language to be transformed, requiring less effort to understand [172]. They are specified at the higher, syntactic level, thereby abstracting from context information that is not relevant for the actual transformation, such as the characters preceding or following the keyword `GO`. A higher abstraction level can also improve comprehension of code and reuse [125], and reduce the chance of errors and improve productivity [49]. This is because the cognitive distance between the task that must be carried out and the actual implementation is often smaller when programming at a higher level. In the Restructuring project from Chapter 5, we perform code restructuring transformations using syntactic tools, resulting in relative simple and intuitive search-and-replace code patterns. A lexical approach for those transformations will result in intricate code patterns that are difficult to comprehend and modify, and they are prone to errors. If we want to implement an analysis to count the forward goto statements, we can easily reuse and adapt the above pattern:

```
count-forward-gotos( Paragraphs Label. Statements. )
=
  count-goto-lab( Paragraphs, Label )
+ count-forward-goto( Paragraphs )
```

However, a syntactic approach for code analysis and modification has quite some challenges. One of the drawbacks is the upfront costs for developing a grammar for the language to be parsed. Also, a new set of programs can break a syntactic tool and require adaptations [24]. Even when a program has been parsed using a grammar, a common problem is the disambiguation of grammars. When more than one parse tree can be constructed from a program, the grammar is *ambiguous*. If, for a particular transformation, an ambiguity arises in a part of the program that is not relevant for the transformation, it is tempting to employ lexical tools for that transformation. Another issue with syntactic tools is the preservation of the original layout and comments. The mechanism to abstract from layout and other lexical information can hinder control over this information, and care should be taken to preserve or reconstruct the original layout and comments.

For these reasons, lexical and syntactic tools are often used in combination with each other. One way is to use syntactic tools for a precise analysis prior to a lexical modification [23]. Such an analysis can yield the exact line number and column number in a program where a change must be made. Using the analysis results, a lexical tool can be used to carry out the actual change. After that, a syntactic tool can be used again to see whether the modified program can still be parsed, and, in addition, a number a postconditions can be checked. This approach has a high accuracy and does not suffer from the layout preservation problems of a syntactic approach. Another way to unite a lexical and syntactic approach is to use lexical tools to build a (partial) parse tree [9, 64]. This overcomes some of the robustness problems of a full syntactic approach, and is suitable for various kinds of analyses. On the other hand, such an approach still suffers from the accuracy limitations of lexical tools and falls short if a high accuracy manipulation is required. A common approach to combine lexical and syntactic tools, originating from compiler construction, is to employ lexical tools to handle preprocessing prior to parsing, and postprocessing to resolve lexical issues afterwards. This is the approach we used in this thesis. In many cases, it is convenient to preprocess source code with a lexical tool before parsing it [41]. Languages often have syntactic freedom that can make parsing hard; some of this freedom can easily be taken away by a preprocessor and results in a simpler grammar. For example, Cobol source lines can start with a line number. These numbers have a very regular pattern: they appear in the first columns of each line. But if the line numbers are incorporated into a grammar, the grammar and transformation patterns become very complex. Instead, a simple lexical tool with a regular expression pattern can easily remove the line numbers before the code is parsed.

So, although lexical tools should be used with care, they are often indispensable to perform a preprocessing step for syntactic tools. It is important for a practitioner to understand the merits and limitations of lexical tools: these tools should be deployed with care and depending on the task that has to be carried out. We illustrate this in the Btrieve project in Chapter 4, where we carry out a full-blown transformation project including code explorations, pre- and postprocessing, grammar engineering and parsing.

In Table 2.1, we summarise a few differences between lexical and syntactic tools. Lexical tools can often be developed faster, operate at a high execution speed, but have a low accuracy and are not very flexible. For a new problem, it may require much effort to adapt an existing tool, or an entirely different one must be developed. On the other hand, syntactic tools operate slower, and can have a longer development time because a suitable grammar is a prerequisite. In order to parse a new set of programs, it can be required that the existing grammar must be adapted, or a new one must be obtained. But once a suitable grammar is acquired, using dedicated technology, one can quickly implement flexible syntactic tools, which allow for accurate analysis and manipulation of source code.

2.2 Generic Language Technology

We illustrated that syntactic tools have several advantages when it comes to performing accurate and versatile code manipulations. Generic language technology is well-suited

Table 2.1: A comparison of lexical and syntactic tools

	lexical	syntactic
development time	++	+
execution speed	++	+
accuracy	-	++
flexibility	-	++
robustness	+	-

to create syntactic tools. This technology is language parameterised, which means that it uses a grammar to generate language specific tools and programming environments. For example, a parser, a syntax highlighter, and a pretty printer can be generated. Since software assets often consist of different languages and dialects, generic techniques are most welcome for analyses or modifications. Therefore, generic language technology is considered to be the basis for automated software maintenance. Examples of generic language technology are the ASF+SDF Meta-Environment [45, 112], the Synthesizer Generator [166, 167], Stratego/XT [110, 208], DMS [15, 16], TXL [62, 63], and Software Refinery [1, 165].

In our research, we used the ASF+SDF Meta-Environment to carry out automated maintenance projects. We describe here how we used the technology for our purposes.

2.2.1 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [45, 48, 112] is a development environment for the automatic generation of interactive systems for manipulating programs, specifications or other texts written in a formal language. SDF stands for Syntax Definition Formalism and supports the definition of both lexical and context-free syntax (production rules), which can be used to specify the syntax of programming languages and program translations. SDF [96] is a modular formalism which contains constructs such as imports, exports and hiding. ASF [21] stands for Algebraic Specification Formalism and supports the definition of equations (conditional rewrite rules). ASF has sufficient expressive power to implement typechecking, program translation, and program execution. Together, ASF+SDF is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. We explain the operation of the ASF+SDF Meta-Environment in this section with elaborate examples; for a thorough description we refer to the documentation [48].

Using an ASF+SDF specification, the ASF+SDF Meta-Environment can generate components for automated software maintenance [40, 44]. In particular, parsers, transformers, unparsers and pretty printers can be generated, which are essential tools for automated software maintenance.

Parsing and disambiguation In order to parse source code, SDF can be used to specify the production rules of a language's grammar. In Figure 2.1, we show an SDF produc-

```
"GO" "TO"? Procedure-name -> Goto-statement
```

Figure 2.1: SDF production rules for the syntax of the goto statement in the Cobol grammar.

```
IDENTIFICATION DIVISION.
PROGRAM-ID COBOLPROGRAM.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 VAR1   PIC 99 VALUE 1.
01 VAR2   PIC 99 VALUE 2.
PROCEDURE DIVISION.
  ADD VAR1 TO VAR2
  DISPLAY VAR2.
```

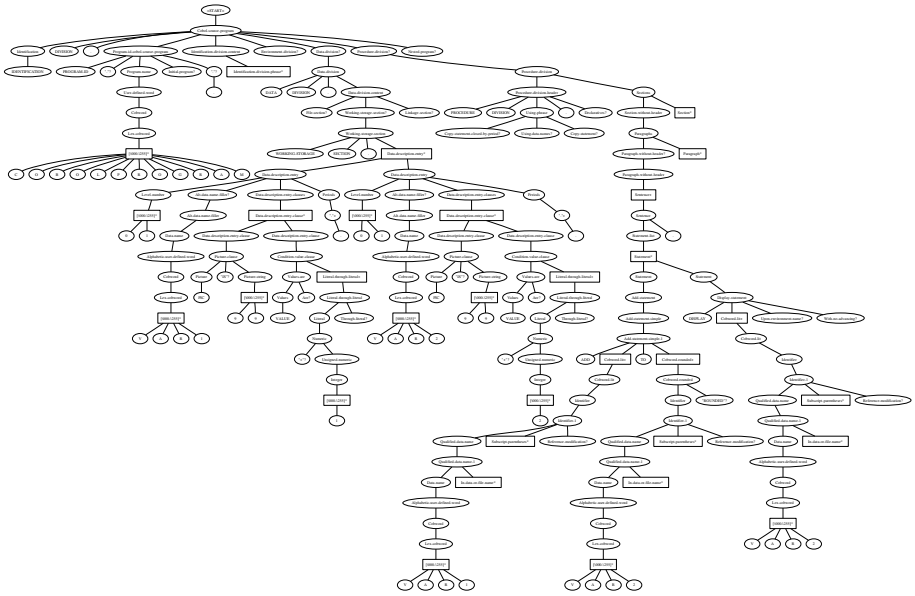

Figure 2.2: A 9 lines Cobol program and its parse tree.

tion rule for the goto statement in Cobol. As opposed to grammar formalisms similar to (Extended) Backus-Naur Form, the resulting non-terminal (or sort) of an SDF rule is on the right-hand side. On the left-hand side, there are the terminals `"GO"` and `"TO"`, and the non-terminal `Procedure-name`. The terminal `"TO"` is optional, indicated by the question mark. When a grammar is specified in SDF, a parse table can be generated. The ASF+SDF Meta-Environment comes with a Generalised LR parser [206], which uses a (generated) parse table for parsing. GLR parsing [133, 191], compared to the traditional LR and LL parsing, is particularly suitable for automated maintenance tasks [43] because it can deal with arbitrary context-free grammars. This means that a grammar can reflect the intended conceptual structure of the language it represents, whereas LR and LL parsing impose certain restrictions on the structure of the production rules, resulting in a more complex grammar. It is also mentioned in [24] that the traditional parsing techniques are not very suitable for automated software maintenance.

The parsing of source code with the GLR parser yields a parse tree [30], representing the original input code. In Figure 2.2, a small Cobol program and its parse tree are depicted. In case more than one parse tree is obtained (a parseforest), the grammar is ambiguous. Disambiguation of grammars is a challenge, and the ASF+SDF Meta-Environment provides several ways to support this [39, 36]. In particular, ambiguities are not resolved by changing the existing production rules of a grammar, but they are treated as a separate concern. Disambiguation is achieved by adding separate restrictions, reject productions, filter attributes, and filtering using term rewriting [36]. This way, the existing production rules remain (mostly) intact.

Transformations When we mention code transformations, we mean both analyses and modifications of source code, i.e. the code of the software artefact to be maintained in the context of automated maintenance. Transformations in the ASF+SDF Meta-Environment are implemented as rewrite rules specified in ASF+SDF. The signature of a transformation is specified as a function in SDF, indicating the (non-)terminal(s) to match (left-hand side) and the resulting non-terminal (right-hand side). Optionally, a visit strategy can be added to a function, which specifies how the function should traverse the parse tree (i.e. in what order the nodes in the tree are visited). In that case, the function is a *traversal* function [35]. Traversal functions implicitly traverse a parse tree and are defined for specific non-terminals by the user; users do not have to implement this visitor behaviour themselves. At each visited node that is of these non-terminals, the rewrite rules for the function are tried. The accompanying ASF rewrite rules have a left-hand side pattern and a right-hand side pattern, and can have conditions that must be satisfied. Patterns are composed of the abstract and concrete syntax of the language to be manipulated, i.e. variables representing syntactic structures are combined with characters and tokens, to form syntactically correct patterns (also called *native patterns* [172]). The variables that are used in a pattern are declared in SDF; hence, they are not variables in the language to be transformed.

We give an illustrative example. In Figure 2.3, the signature of a traversal function `count-gotos()` and several variable declarations are specified in SDF, and the behaviour of the function is implemented in an ASF rewrite rule. The function takes a Cobol program or a goto statement and an integer as input, and returns the number of goto

```
%% SDF function declarations
  count-gotos ( Cobol-source-program , Integer )
              -> Integer {traversal(accu,bottom-up,continue)}
  count-gotos ( Goto-statement       , Integer )
              -> Integer {traversal(accu,bottom-up,continue)}

%% SDF variable declarations
  "TO?"                          -> "TO"?
  "Procedure-name"               -> Procedure-name
  "Integer"                      -> Integer
```

```
%% ASF rewrite rule
  [tag] count-gotos( GO TO? Procedure-name , Integer )
                   =  Integer + 1
```

Figure 2.3: The syntax and semantics of the `count-gotos()` function specified in ASF+SDF.

statements. The traversal type and strategy for visiting the parse tree is also provided: `traversal(accu,bottom-up,continue)`. The keyword `accu` stands for *accumulator*, which means that the function gathers information from a parse tree (as opposed to manipulating a parse tree). In this case, an integer value indicates the number of goto statements. The information is stored in the second argument of the function, acting as a global, modifiable state during the traversal. Each time a goto statement is visited, the integer is updated and implicitly passed on. The keywords `bottom-up` and `continue` specify the visiting strategy, indicating that the nodes in the parse tree are visited in a *bottom-up* fashion (as opposed to top-down), and after a particular node has been transformed, the traversal *continues* with its ancestors (as opposed to stopping at that point). The visiting strategies will be discussed in more detail later. We mention that for this accumulating transformation the order of visiting the nodes is irrelevant: we need to visit all goto statements but in no particular order. Then, three variables are declared, which are used in the ASF rewrite rule at the bottom of the figure. The first argument in the left-hand side pattern of the rule represents the syntax of the goto statement (as in Figure 2.1), but now `TO?` and `Procedure-name` are SDF variables. The variable `Integer` represents the accumulated value, and is incremented on the right-hand side of the rule.

Now that the syntax and semantics of the function `count-gotos()` are specified, it can be applied to (parsed) source code. In Figure 2.4, the function is applied to a sample Cobol program. The integer value is initialised with value 0 by the user (one can also implement an additional function that rewrites a program `P` to `count-gotos(P,0)`). If we apply the rewrite engine of the ASF+SDF Meta-Environment, the program is rewritten to an integer with value 1.

As we showed in the example, the ASF+SDF Meta-Environment provides very convenient support for generic traversal [35], which makes it possible to rewrite complex parse trees with concise rules. Traversal functions were first used with the ASF+SDF Meta-Environment in [40, 44] to reduce the manual effort to write renovation transformations. This turned out to be a powerful asset and was incorporated into the ASF+SDF Meta-En-

```
count-gotos(
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COUNTGOTOS.
 PROCEDURE DIVISION.
 LABEL1.
     GO TO LABEL2.
 LABEL2.
     EXIT PROGRAM.
, 0 )
```

Figure 2.4: Application of the function `count-gotos()` on a sample Cobol program.

vironment. To illustrate this, in [41, p 9] a tool is mentioned for adding `END-IF` keywords to if statements in Cobol, comprising 25 ASF rewrite rules. Using traversal functions, this tool can be implemented with only a few rules [44].

Three types of traversals are supported: an accumulator (which we showed above), a transformer (a sort-preserving transformation), and an accumulating transformer (a combination of the two). The accumulator is useful for doing analyses, since it collects information from a parse tree. A transformer is useful for modification of source code; it can take a program as input and returns a transformed program. An accumulating transformer traverses a parse tree, thereby collecting information and also modifying the parse tree. This can be used to concisely implement a transformation that needs specific context information when making a change, i.e. information on earlier changes made by that transformation. For example, a transformation that labels each statement with a unique number or identifier needs to know which numbers or identifiers have already been used.

Each type of traversal can be implemented with a visiting strategy, determining the order of the traversal and its behaviour after a matching node has been found. The visiting strategy can be *bottom-up* or *top-down*, and can have an additional attribute for defining the continuation behaviour. There are two attributes: *continue* and *break*. The default behaviour is to continue, which means that the traversal visits all nodes of the specified sort in a tree. However, this may not always be desirable. The break attribute allows for control over a traversal, because it permits a user to stop or continue the traversal under certain conditions. With two visiting strategies and two attributes, there are four combinations possible for a traversal: {top-down,break}, {top-down,continue}, {bottom-up,break}, and {bottom-up,continue}. Depending on the transformation that must be carried out, one can choose a suitable strategy.

In general, if explicit control over a traversal is required then one uses a strategy with a break attribute, and otherwise one uses a continue attribute. The result of a transformation can depend heavily on the chosen visiting strategy. In particular, one has to be careful when transforming nested structures, such as statements containing other statements. A simple example is given in Figure 2.5, where an ASF rule for counting statements is depicted. Depending on the visiting strategy, this rule has different results: any strategy with a continue attribute counts *all* statements in a program, a {top-down,break} strategy skips statements *contained* in other statements (after matching the containing statement, the traversal does not traverse further down to the children of the current node), and a

```
[tag] count-statements( Statement , Integer ) = Integer + 1
```

Figure 2.5: An ASF rewrite rule for counting statements. The result of this transformation depends heavily on the chosen visiting strategy.

{bottom-up,break} strategy skips statements *containing* other statements (after matching a contained statement, the traversal does not traverse further up to the ancestors of the current node). The example illustrated the versatility of the traversal function, allowing one to concisely implement refined transformations, such as counting the number of contained or containing statements. We found that the supported strategies were adequate to concisely implement the transformations in this thesis.

Unparsing and pretty printers The ASF+SDF Meta-Environment provides an unparser to translate parse trees to text, as well as support for generic pretty printing [47]. The notions of unparser and pretty printer are often used interchangeably. In the ASF+SDF Meta-Environment, pretty printing is regarded as formatting a parse tree according to a specific convention (it is in fact a transformation), and unparsing as translating a parse tree to text. The formatting capabilities of the ASF+SDF Meta-Environment are implemented by a multistage formatting pipeline, which allows one to express *user-defined* context-sensitive formatting [37, 38], combined with *default* formatting. This means that there are a number of default formatting rules (e.g. a list of statements is formatted vertically, a statement is formatted horizontally when there is enough space), which can be completed or overruled by user-defined rules using context-information (e.g. apply specific indentation only in a certain context). This allows one to implement corporate coding standards that cannot be enforced by off-the-shelf formatters.

Formatting rules are expressed in a similar way as a transformation: by using rewrite rules. A special language, called Box [47], is used to describe the desired layout using a number of operators. For instance, an `H` operator formats horizontally, whereas a `V` operator formats vertically. Each operator has parameters to fine-tune it; for example, `I is=4` defines an **i**ndentation **s**pace of 4. In a user-defined rule, the constructs of the language to be formatted are explicitly mapped on Box operators. Therefore, Box syntax must be declared for those constructs. This is similar to a declaring a function for a transformation, but the Box declarations have a special attribute: `from-box` or `to-box`. After the transformation to Box syntax, a dedicated tool formats the code using the from-box and to-box information in the parse tree. We show an example in Figure 2.6, where a formatting rule for the goto statement is shown. The additional Box syntax is declared in SDF, as well as the variables for the ASF rewrite rule. The rewrite rule itself specifies the desired formatting for the goto statement: the statement is printed horizontally (`H[]`), and the variable `Procedure-name` is printed at **t**ab **s**pace 15 (`H ts=15[]`). This means that a procedure name that appears in a goto statement will be printed at column 15, provided that there is enough space. The number of spaces between the `GO` and optional `TO` is not specified, so the default space of the `H` operator is taken, which is one position. The variables in `to-box` operators are formatted by either the default rules or another user-defined rule, if one is implemented. In Figure 2.7, we show a code fragment with goto statements before and after formatting. The fragment illustrates that the formatting

```
%% SDF syntax declaration for Box
  from-box( Box )              -> Goto-statement {from-box}
  to-box( "TO"? )              -> Box {to-box}
  to-box( Procedure-name )     -> Box {to-box}

%% SDF variable declarations
  "TO?"                        -> "TO"?
  "Procedure-name"             -> Procedure-name
```

```
%% ASF rewrite rule
   [tag]
   GO TO? Procedure-name
   =
   from-box(
     H [ "GO" to-box(TO?) H ts=15 [ to-box(Procedure-name) ]]
   )
```

Figure 2.6: The specification for formatting the goto statement, implemented as an ASF rewrite rule.

```
* before formatting          | * after formatting
IF A > B                     | IF A > B
  GO TO LABEL1               |   GO TO       LABEL1
END-IF                       | END-IF
GO TO LABEL2                 | GO TO         LABEL2
```

Figure 2.7: Formatting of (nested) goto statements using the formatting transformation rule from Figure 2.6.

rule enforces an alignment of the labels in the goto statement, regardless the indentation of the statement itself. For more complex formatting rules using context information, and for more formatting examples, we refer to [37, 38].

Layout and comment preservation Common problems in automatic modification of source code are the preservation of layout and, in particular, comments. In many languages, comments may appear in arbitrary places. A grammar and transformation that take comments into account are more complex, because there is a larger cognitive distance between the grammar and the language to be parsed. Therefore, comments are often treated as layout, which means that they can appear anywhere in the input but do not appear in the grammar and transformation rules. However, layout is often lost or malformed during a transformation (e.g. when a statement is removed or replaced). Although layout such as spaces and newlines can be reformatted using a pretty printer, this is not possible for comments. The current implementation of the ASF+SDF Meta-Environment preserves layout in places that are not rewritten [46], but that is not always sufficient for automated maintenance tasks. Work is under way to improve the preservation of layout and thus comments in the ASF+SDF Meta-Environment [204, ch. 8]. In the Btrieve

```
%% case-sensitive keywords in SDF
  "GO" "TO"? Procedure-name          -> Goto-statement

%% case-insensitive keywords in SDF
  Go-KW To-KW? Procedure-name        -> Goto-statement

  [Gg][Oo]                           -> Go-KW
  [Tt][Oo]                           -> To-KW
```

Figure 2.8: Case-sensitive and case-insensitive goto statements specified in SDF production rules. The case-insensitive variant requires additional rules for the keywords.

project in Chapter 4, we inserted comments into the layout, whereas in the Restructuring project in Chapter 5, we added comments explicitly to a grammar and the accompanying transformation rules; this allowed for full control over the existing comments. It turned out that it is not feasible to automatically preserve and transform all comments as they were originally intended.

Case-sensitivity Many programming languages are case-insensitive. The current implementation of the ASF+SDF Meta-Environment does not support case-insensitivity. Hence, case-insensitivity must be specified explicitly in the grammar before parsing, resulting in a more complex grammar [41]. This is illustrated in Figure 2.8, where case-sensitive and case-insensitive production rules for the goto statement are presented. The case-insensitive grammar requires additional productions. A different option is to preprocess the source code before parsing, such that all keyword characters are changed to the same case. From a technical point of view, this option is more attractive. If all keywords are all either lowercase or uppercase, this solution works well. All keywords are changed to the desired case by preprocessing, and changed back after the transformation. However, if lowercase and uppercase are used interchangeably, it is difficult to restore the original case of a character. In the Btrieve project in Chapter 4, we changed all keywords to uppercase prior to parsing, and changed them back afterwards. We have been told by one of the developers of the ASF+SDF Meta-Environment [203] that work is underway to support case-insensitivity.

2.2.2 Other GLT systems

We mentioned other GLT systems: the Synthesizer Generator [166, 167], DMS [15, 16], TXL [62, 63], Stratego/XT [110, 208], and Software Refinery [165]. For a brief list of programming language tools, see [97]. These systems can be used for automated maintenance tasks, taking a language's specification as input and generating a language specific maintenance tool.

The systems differ in several aspects: language formalism, parsing technology, transformation strategies, availability, et cetera. We do not discuss the systems in detail, but briefly review some differences and similarites of these systems with the ASF+SDF Meta-Environment, which are relevant for automated maintenance tasks. We also provide some

pointers for further information.

Availability The Synthesizer Generator [92], DMS [175], and Software Refinery [165] are commercial products. For comparisons of Software Refinery and the ASF+SDF Meta-Environment, see [44, 69]. TXL [192], Stratego/XT [188], and the ASF+SDF Meta-Environment [48] can be obtained for free.

Parsing Stratego/XT and the ASF+SDF Meta-Environment use the same language formalism (SDF) and GLR parsing technology; DMS also uses GLR parsing. TXL, the Software Refinery and the Synthesizer Generator use (LA)LR parsing. TXL allows for some form of automatic grammar modification by redefining production rules, and allows to switch to case-insensitive parsing of the input. For a discussion on parsing technologies for automated maintenance, see [43].

Transformation In most of the systems, transformations are implemented as rules with a left-hand side pattern and a right-hand side pattern, and optionally a condition that must be satisfied in order to trigger a rule. The application order of a rule can often be configured by programming a strategy, similar to the traversal strategies we discussed.

Formatting In TXL and DMS, formatting rules can be embedded in the production rules of the grammar, whereas in Stratego/XT and the ASF+SDF Meta-Environment formatting rules are specified separately. Stratego/XT and the ASF+SDF Meta-Environment use the Box language for specifying formatting rules, and DMS uses a Box-like language.

2.2.3 Grammar engineering

We explained that a key to success for automated maintenance are syntactic tools, and that generic language technology can aid the development of such tools. However, a suitable grammar is required as an input for the generic language technology. We briefly touch upon the notion of a grammar in the context of a programming language, and then discuss engineering of grammars.

Grammars In the mathematical sense, a grammar G is defined as a quadruple $G = (V, T, S, P)$, where V is a set of variables, or non-terminals (sorts), T is a set of terminals, S is the start-symbol, and P is a set of production rules [144]. In a context-free grammar, which we consider here, all productions have the form $A \rightarrow x$, where $A \in V$ and $x \in (V \cup T)*$. Terminals, or tokens, consist of one or more actual characters of the language represented by the grammar, and in a programming language, they usually represent keywords, identifiers, constants, operators, and so on. Non-terminals are constructed by grouping terminals and other non-terminals according to the production rules, representing expressions, statements, functions, and so on. The start-symbol is a special non-terminal, which is the top level non-terminal, and it is usually an entire program. The production rules form the heart of the grammar, showing how the non-terminals are

constructed from other non-terminals and terminals, using a left-hand side and right-hand side.

There are several notations for production rules, but the most well-know are probably Backus-Naur Form (BNF) and Extended BNF. In this thesis, we mainly use the notation of the Syntax Definition Formalism (SDF) [96]. We already showed several examples of SDF when we discussed the ASF+SDF Meta-Environment. The most significant difference between (E)BNF and SDF in writing production rules is that the left-hand side and right-hand side are swapped. In (E)BNF, the resulting non-terminal of a rule is written on the left-hand side, whereas in SDF it is written on the right-hand side (which is the opposite notation of the formal definition we gave earlier):

```
A ::= B "c" D     % BNF
B "c" D -> A      % SDF
```

In addition, SDF provides an integrated definition of lexical syntax (terminals) and context-free syntax (non-terminals), and an integrated way of defining associativity and priorities for production rules.

Grammars are often constructed and adapted in an ad hoc way [24, 43], but effort is put into turning this into a discipline [113]. There, the notion of grammarware is introduced, which comprises grammars and all grammar-dependent software, including tools for automated software maintenance. We refer to that paper for an elaborate overview and references to work related to grammars and grammarware.

Obtaining and deploying a grammar One way to obtain a grammar for an existing language is to construct one incrementally by parsing a set of source files. With this approach, it is very likely that the obtained grammar has to be customised for other source files, because not all syntactic constructs are covered by the first set of files. A more structured approach is described in [131], where the authors propose to recover grammars semi-automatically from language references, compilers, and other artefacts. The approach was used to obtain an IBM VS Cobol II grammar, which can be found online [129]. That recovered grammar was also used as a basis for the Cobol projects in this thesis.

Still, it is not true that a recovered grammar can be used to parse any source file in that language; real-world code usually deviates from the syntax in a reference manual (e.g. ill-documented syntactic freedom has been used), and such a grammar is often ambiguous [131]. Therefore, an automatically recovered grammar also has to be customised before it is suitable for deployment, i.e. parsing of source code. First of all, the grammar must be suited for the formalism of the parser generator. For example, the parser generator Yacc uses BNF, which imposes restrictions on the structure of the production rules. This requires *yaccification* [131, 132]: elimination of optionals, lists, and nested alternatives. In more versatile formalisms supporting these operators, such as EBNF and SDF, this is not required. Furthermore, a grammar is often ambiguous. Depending on the used grammar formalism and parser generator, different disambiguation methods are available. When using Yacc for generating a parser, ambiguities (or conflicts) are resolved by user-controlled precedence and default behaviour. The default behaviour can be changed by

```
goto-statement
  : "GO" "TO"? procedure-name
  ;
```

Figure 2.9: The Cobol goto statement in the LLL grammar formalism.

```
%include goto-statement
  :  "GO" "TO"? procedure-name+ "DEPENDING" "ON"? identifier
  ;
```

Figure 2.10: A GDK grammar transformation to include an alternative for the Cobol goto statement.

modifying the order of the production rules. In the ASF+SDF Meta-Environment, disambiguation is seen as a separate concern to keep the impact on the existing production rules as low as possible. All possible parse trees of an input text are generated, resulting in a parseforest. User-controlled strategies are applied to select the correct parse tree: separate restrictions, reject productions, filter attributes, and filtering using term rewriting can be implemented to remove incorrect parse trees from a parseforest. In [131], it is discussed how the recovered IBM VS Cobol II grammar was disambiguated, and in [36], examples of filtering using term rewriting are presented.

Tool support for grammar deployment The Grammar Deployment Kit (GDK) [122] provides support for grammar deployment, i.e. grammar adaptation [126] and parser generation. GDK takes a grammar specification in an ENBF-based grammar format, allows for several grammar transformations, and can generate parsers. In addition, a library is provided which allows to develop simple software transformation tools.

GDK uses the grammar formalism LLL, which stands for Lightweight LL parsing (the built-in parsing technology of GDK). In Figure 2.9, the goto statement in Cobol is shown in LLL. A number of operators for grammar adaptation are provided, which allow rules and non-terminals to be added, modified or removed. A grammar transformation is specified in a transformation script, which enables one to keep track of the grammar changes. This way, changes are not recorded in the grammar itself but in a separate file. Moreover, comments can be added to document the changes. To illustrate a grammar change, we extend a grammar rule with another alternative, using the `%include` operator. In Cobol, besides the usual goto statement with a single label, there is also a goto statement with multiple labels. A label is chosen depending on the value of an identifier. To extend the goto statement from Figure 2.9 with this type, we specify a grammar transformation in Figure 2.10 which includes an alternative to the existing rule. The result is shown in Figure 2.11. In a similar way, `%exclude` can be used to remove alternatives. Furthermore, various operations can be applied to individual non-terminals or entire rules.

GDK can generate parsers from LLL grammars, supporting various technologies such as BtYacc (Yacc with backtracking functionality), SDF, and its built-in parsing technology. In fact, GDK can generate parser input specifications, including code for parse tree construction, scanner templates and a simplified form of rewriting. However, since sev-

```
goto-statement
  : "GO" "TO"? procedure-name
  | "GO" "TO"? procedure-name+ "DEPENDING" "ON"? identifier
  ;
```

Figure 2.11: The modified goto statement in LLL.

eral parser technologies use specific ways for disambiguating a grammar (e.g. by adding attributes to rules), a generated grammar specification for a specific technology usually requires additional modification.

Parsing *or* pre- and postprocessing Many languages have a certain syntactic freedom that can complicate grammar specifications significantly. For example, a grammar becomes very complex by incorporating source line numbers, optional syntax constructs, mixed case keywords, and comments, since these can appear virtually anywhere in source code. This is discussed for Cobol in [41]. Another issue that can complicate parsing are preprocessor statements, because in several languages unpreprocessed code does not have to be syntactically correct. The C/C++ preprocessor directives and its use in industrial systems is particularly well-known to complicate automated maintenance tasks [17, 90, 209]. In addition, if irregular syntax is embodied into a grammar, it is very likely that the accompanying transformation rules become complex as well.

In many cases, these problems can mostly be tackled by using a tailor-made preprocessor that massages the input code. This prevents the grammar from becoming too complex [41]. A postprocessor reverts the lexical modifications after unparsing. Pre- and postprocessors can be implemented using lexical technology like Perl and sed & awk. One has to decide what should be incorporated into the grammar and what should be handled by the preprocessor; this decision can also depend on the transformation that must be carried out. If a transformation needs to analyse information from include files, it can be convenient to expand the include files before parsing. However, since a lexical tool is only suitable for simple text replacements, its capabilities are limited. Moreover, when doing automated code transformations, it is often desired to undo preprocessor changes afterwards using the postprocessor; this should also be taken into account when designing a preprocessor. For example, include files, expanded by a preprocessor, must be collapsed by the postprocessor. Scaffolding techniques [174] can be useful here: extend a grammar slightly at designated points or use existing production rules to store additional information, which can be used at a later moment.

Tolerant grammars There are many occasions when one is interested in only a subset of the syntactic constructs of a language, while disregarding syntactic irregularities in other constructs. Then, a *tolerant* grammar can bring relief [116].

We discussed earlier that a lexical approach is only suitable for certain situations, and that a syntactic (or context-free) approach is more precise and allows for in-depth syntactic and semantic analyses. However, a normal parser with a full grammar has upfront investment and practical limitations (e.g. erroneous input files, embedded languages, different dialects). This has resulted in tolerant parsers using so-called *island* grammars [155, 202].

An island grammar describes constructs of interest as islands, while the rest of a language is considered as water. This way, a parser is not obstructed by irregularities in constructs that are not of interest.

Still, island grammars can have severe limitations, as illustrated in [116]: because islands are not connected to each other, i.e. there is no structure indicating their relation, island grammars have a chance of false negatives. In addition, the definition of the islands can become too tolerant (as shown in [116]), which can cause false positives. So, if high accuracy is required, island grammars may not be sufficient. A slightly different approach is to use a *skeleton* grammar, that is, a structured tolerant grammar. To be more precise, a tolerant grammar and its base-line grammar (the full grammar) share the same production rules on the path from the start-symbol to the constructs of interest. Production rules that are beyond this set of rules are replaced by default productions. This way, the tolerant grammar has enough structure to avoid false positives and false negatives, if it is assumed that the base-line grammar is complete and correct. A formal definition of tolerant grammars, together with an example derivation of a tolerant grammar, is given in [116]. Skeleton grammars were used for the research in this thesis.

2.3 Putting it together: Software maintenance factories

We have discussed several tools, and we showed what they can do to support automated maintenance tasks. We now summarise how these tools can be combined in an infrastructure, which we used in several projects in this thesis: software maintenance factories. An architecture for a software maintenance factory, generation of factory components, and examples of factories are described in [44, 52, 173, 199]. In this section, we show what our software maintenance factories look like, and how the tools we have just discussed are related to a factory's operation.

In Figure 2.12, we give an overview of a software maintenance factory and how its components are created. Source code to be analysed or manipulated enters the assembly-line of the factory, and the processed code is the output at the bottom of the figure. The factory has five phases: preprocessing, parsing, transformation, unparsing and postprocessing. A transformation phase usually consists of several smaller transformations that are combined. The tools we described in this chapter are depicted in shaded boxes, and allow us to implement the components that are used in the different phases. The pre- and postprocessors are implemented in Perl, grammar specifications are obtained and deployed using GDK (assuming there is a base-line grammar), and the ASF+SDF Meta-Environment is used to generate parsers, implement transformations and provide an unparser.

For each specific maintenance task, a software maintenance factory can be constructed. Fortunately, many of the factory components from Figure 2.12 can be reused or generated [44]. For instance, a tailor-made pre- and postprocessor for Cobol can be deployed in another factory, possibly adapting it to the problem at hand. The parser, transformation, and unparser components are generated using generic language technology, in our case the ASF+SDF Meta-Environment. This way, a new factory can be constructed quickly.

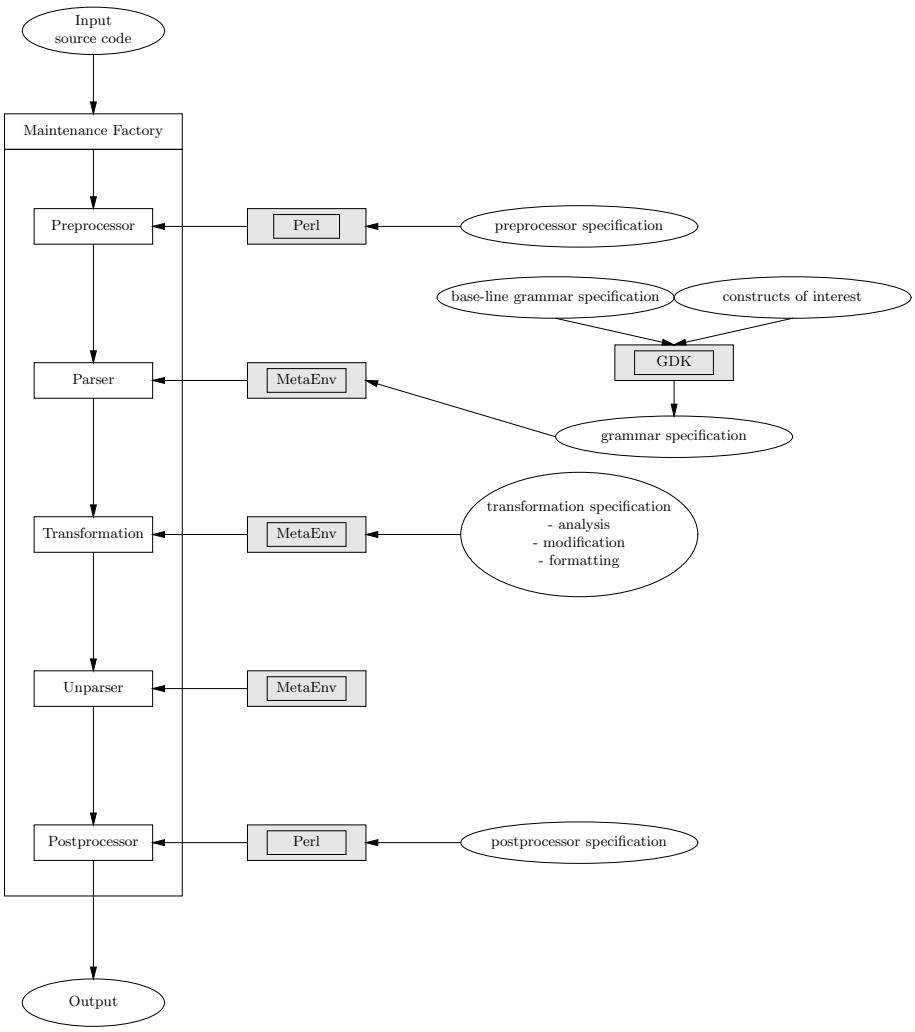

Figure 2.12: The architecture of our software maintenance factories and technology to create their components.

2.4 Summary

In this chapter, we presented building blocks and how they aggregate into maintenance factories, which are suitable to carry out massive maintenance. We have explained the advantages of lexical and syntactic tools for automated software maintenance. We illustrated that lexical tools are suitable for simple tasks, such as a quick code analysis or the pre- and postprocessing of source code. Despite the upfront costs for parser development, syntactic tools should be preferred for accurate and versatile analyses and manipulation of software.

Syntactic tools can be created using generic language technology, that is, language parameterised technology. The technology takes a grammar as input and can generate language specific tools like a parser. We explained the generic language technology that we used for the projects in this thesis: the ASF+SDF Meta-Environment. It provides versatile support for parsing and transformations. We also mentioned some issues that should be considered when using generic language technology, such as the preservation of comments in transformed code.

A prerequisite for generic language technology is a suitable grammar specification. We explained how a grammar can be obtained and customised to be suitable for deployment, and we illustrated the grammar deployment tool GDK. We also explained that it is often more convenient to preprocess some of a language's syntactic freedom instead of incorporating it into a grammar. Furthermore, we discussed tolerant grammars, which are created by using a subset of a language's grammar.

Finally, we showed how all tools can be put together in software maintenance factories. A software maintenance factory consists of several components, which are arranged in an assembly-line. The factories that we implemented consisted of several components: preprocessor, parser, transformation, unparser, and postprocessor. Source code enters a factory and is processed on the assembly-line by each component, and at the end of the assembly-line the processed code leaves the factory. Most of the components can be generated and/or reused.

More details on the architecture and implementation of our tools are given throughout the thesis.

Road map Now that we have discussed mass maintenance tools and technology, we are ready to present several chapters where we developed and deployed them in industrial settings. Starting with the Minefield project in Chapter 3, we work on the analysis of coding standard violations and the automatic transformation of complex source code.

Chapter 3

Cobol minefield detection

In Cobol, procedures can be programmed in ways that lead to unexpected behaviour and reduced portability. This behaviour is recognised as so-called 'mines': programming practices containing hidden dangers and requiring extreme caution. Cobol mines can be created intentionally or by a programming error, and can be tripped at an unforeseen moment. This leads to minefields in source code with unseen hazards, which complicate understanding and maintenance, and which can lead to costly breakdowns of business-critical software systems. Cobol minefields are often the result of software evolution.

In this chapter, we discuss Cobol mines and the dangers that come with them, and we implemented a mine detector for Cobol. Our detector was deployed in an industrial case: the Minefield project. We found several minefields in production systems. By restructuring an evolved legacy application, we argue that code restructuring can be used to combat minefields. This chapter is based on: N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice & Experience*, 36(14):1605-1642, 2006 [197].

3.1 Introduction

Programming errors are present in every software system, and are more than once a source of trouble. Errors can lay dormant in an application, ready to show up at any moment in time. When a dormant error arises during production, the system can suddenly break down. The origin of the error is often hard to find, causing high expenses. Sometimes, a programmer uses an awkward programming style intentionally to implement certain logic. Reasons for this can be, for instance, the absence of a proper language construct, or a preferred programming style. Such programming practice can eventually lead to problems when another programmer, who is not familiar with such logic, has to modify the code. Or when the code is migrated to a different environment, the behaviour of the application can become entirely different.

The use of coding standards can aid to prevent awkward programming styles. For example, a common coding standard in Cobol requires that dependencies between procedures are indicated by an alphabetic and numeric hierarchy of their names. All procedures appearing at the same level start with the same letter. This means that the main procedure

in a program starts with `A1_MAIN` and all procedures that are directly referenced from that procedure start with `B01_..`, `B02_..`, and so on. Procedures which are approached from several different levels often fulfill common tasks such as error handling, and start therefore with a letter from the end of the alphabet (e.g. `Y01_LOG_ERROR`). A coding standard is often developed at the initial stage of the coding process, and is therefore hardwired into the code; it is part of the applications' architecture [117, 169]. Consequently, coding standards can be regarded as *source code level architectural decisions*, guarding the quality of the source code. A violation of a standard may indicate a programming error, or potential programming malpractice causing error-prone code. With the help of analysis tools, one can detect intentional and unintentional violations of a standard, which can then be tracked down, mapped, and possibly repaired before a crash appears during a production run.

In this chapter, we investigate the phenomenon 'mines' in Cobol. Mines represent potential hazardous programming constructs. In an industrial setting, we implemented a mine detector for Cobol and applied it to several systems. Furthermore, we show how code restructuring can demarcate Cobol minefields and other complicated code in a program.

Contributions This chapter has several contributions. First, we discuss different implementations of the Cobol perform statement (Cobol procedure call), and we illustrate the variation in compilers by example programs. This confirms that there is no single semantics for Cobol, and that problems can arise when code must be migrated. Second, we present definitions for potential hazardous, error-prone, and incompatible programming practices in Cobol: Cobol mines. These mines can pose severe threats to the proper operation of a system. Third, we present a case study on Cobol minefield detection, which includes an analysis of five industrial Cobol systems from different companies and an evaluation of the results. Finally, we discuss the demarcation of minefields, which is illustrated by the restructuring and visualisation of an evolved legacy application.

Related work The notion of mines in Cobol is mentioned in [81]. In that paper, an approach is described to identify procedural structures in Cobol programs. An elaborate treatment on the behaviour of perform statements is given, and an algorithm is presented to identify non-structured ranges of perform statements, that is, ranges containing a mine. In this chapter, we elaborate on different types of mines, the (possible) consequences of a mine, and how one can fight minefields. Although the authors of [81] recognise that the semantics of the perform statement is not precisely defined, they believe that the implementation of the perform statement in the IBM Cobol compiler [99] is similar to other modern Cobol compilers. According to our findings, this is not the case. We show that the semantics of a perform statement is not fully defined and we demonstrate differences between various compilers.

In [14], a formal semantic description of control constructs of Cobol is given. Since the semantics of a perform statement is not completely defined by the latest Cobol standard, the description holds only for a certain compiler, configuration and environment. In our analysis, we also chose a certain semantics for the perform statement, and elaborate on its consequences for the detection of Cobol mines.

In [71], a case study on the analysis and visualisation of the control-flow of Cobol is presented. Although they elaborate on the flow of control in Cobol, they do not deal with the goto statement or the so-called fallthrough logic. These language features are very prominent characteristics for the control-flow in Cobol and the creation of error-prone code, such as mines. In this chapter, we do take these control-flow features into account while detecting hazardous code.

There are a number of commercial tools for analysing Cobol programs, but we could not find one that suited our projects' needs. Especially an analysis of the fallthrough logic between Cobol paragraphs is not available in many tools. IBM's Cobol Structuring Facility has capabilities to detect intricate code, including several of the mines we define in this chapter, but the tool appears to be no longer in service, and we could not find any case studies that used it for mine detection.

The classification we use for certain hazardous programming practices are quite specific for the Cobol language. Similar problems appear in other languages. For example, the existence of side effects in the C language that have implementer defined semantics [111] can cause portability problems. Another related topic is code smells [84]. Code smells are a kind of anti-patterns for programming [51], indicating that the structure and comprehension of a particular piece of code may be improved by refactoring. A number of code smells are proposed in [84]; some smells are specific to object-oriented programming, but there are also smells that can be applied to other paradigms. The relation to Cobol mines is that both smells and mines can be an indication of error-prone code.

3.2 Cobol control-flow and Cobol mines

3.2.1 Control-flow revisited

In order to understand the control-flow in Cobol, a few things need to be explained first. A Cobol program consists of four divisions: one for a description of the program, one for external dependencies, one for variable declarations, and one for the programming logic. We focus on this last division, which is called the `PROCEDURE DIVISION` and is ordered similar to a text in natural language, using sections, paragraphs and sentences. A section is divided into paragraphs, and a paragraph consists of sentences. A sentence is a sequence of statements, ended by a period. Sections and paragraphs usually start with a label, which can be used to reference them from elsewhere in the program.

When a program is executed, control-flow starts with the first statement in the `PROCEDURE DIVISION`. As soon as the control-flow reaches the end of the last statement in the program, the execution ends. There are also a number of statements that end a program immediately: the `STOP RUN`, `GOBACK` and `EXIT PROGRAM` statements. The precise operation of these statements depends on whether they are executed in a main program or in a subprogram. The `EXIT PROGRAM` statement should not be confused with the plain `EXIT` statement, which has no effect on the execution of a program.

There are a number of constructs that control the program flow. The most famous control-flow construct in Cobol is probably the unconditional `GO TO` statement [73, 168], which jumps to a location indicated by a label. In addition, there is a conditional `GO TO` statement, which jumps to a certain label depending on a condition. This construct is

```
PROCEDURE DIVISION.
MAIN SECTION.
LABEL1.
    PERFORM LABEL2 THRU LABEL-EXIT
    STOP RUN.
LABEL2.
    DISPLAY 'EXECUTING LABEL2'
    IF FLAG
      GO TO LABEL-EXIT
    ELSE
      DISPLAY 'FLAG NOT SET'
    END-IF.
LABEL-EXIT.
    EXIT.
```

Figure 3.1: A Cobol procedure division.

used to simulate a case-like structure. A `GO TO`-related statement is the `ALTER` statement, which modifies the destination of a special `GO TO` statement at run-time. Because the `ALTER` statement can lead to very incomprehensive programs, it has been removed from the Cobol standard. Nevertheless, the `ALTER` statement still occurs in production code. Furthermore, a `GO TO`-like statement is the `NEXT SENTENCE` statement, which transfers control to the next sentence in the code; since a period indicates the end of the current sentence, the `NEXT SENTENCE` statement can lead to programming errors when a period is removed or added without proper care. An example of this problem will be shown in Chapter 5. The `NEXT SENTENCE` statement is commonly used in production systems, but is considered archaic in the latest Cobol standard.

A procedure call in Cobol can be made using a `PERFORM` statement, which is used to execute a sequence of paragraphs or sections. In Figure 3.1, a `PERFORM` statement in `LABEL1` executes a sequence of paragraphs. From `LABEL2`, the control-flow can jump to `LABEL-EXIT` by the explicit `GO TO LABEL-EXIT` statement, or control can continue to `LABEL-EXIT` at the end of `LABEL2` by implicit fallthrough. The `EXIT` statement is purely for the programmer to indicate the end of a sequence; it does not affect the execution of the program. After `LABEL-EXIT`, the control-flow is transferred to the `STOP RUN` statement in `LABEL1`, and the program terminates. We emphasise here that implicit fallthrough between paragraphs and sections can severely complicate program understanding, since such behaviour cannot be seen from a paragraph or section itself; it depends on the way a paragraph is reached. Furthermore, control-flow can be affected by Cobol's exception handling mechanism for files and the possibility to execute statements from a `SORT` statement or a `MERGE` statement; the operation of these mechanisms resembles a `PERFORM` statement, but we do not consider them in this chapter. So, there are basically three ways to transfer the control-flow in a program: by a `GO TO` statement, by a `PERFORM` statement, and by the implicit fallthrough logic.

Then there are several statements which can contain other statements, such as the `IF`, `EVALUATE` (a case-like construct), and the in-line `PERFORM` (a while-like construct). In

addition to these well-know types of branching statements, there are several other statements that allow conditional execution of statements. This can be when an error arises from an operation, or, on the contrary, when no error arises. For instance, an arithmetic error can arise from an addition or division. A simple example of this mechanism is the following `ADD` statement:

```
ADD A TO B
   ON SIZE ERROR      PERFORM OVERFLOW-ERROR
   NOT ON SIZE ERROR  PERFORM COMPUTATION
END-ADD
```

In case variable `B` is unable to store the value of variable `A` (e.g. the value of `A` is too big for `B`), an error routine is performed. If no error occurs, a computation routine is executed. Such structured statements are very similar to an `IF` statement: a condition is evaluated and then one of the branches is executed.

The Cobol 2002 standard [6] extended the Cobol language with a number of constructs that are also available in several newer languages. Important extensions for the flow of control are the new variants of the `EXIT` statement, which transfer control to the beginning or the end of a loop (`EXIT PERFORM`), the end of a paragraph (`EXIT PARAGRAPH`), or to the end of a section (`EXIT SECTION`). These variants resemble statements like the continue, break and return statements which are common in languages like C and Java. In addition, object orientation [213] is supported and exception handling has been improved, but we do not consider those extensions in this chapter. The majority of the production code does not make use of these new features, and, as this chapter is about minefield detection in existing systems, this restriction does not limit the applicability of our approach.

3.2.2 Some semantics of perform statements

The semantics of perform statements differs between Cobol dialects. In particular, the Cobol 2002 standard [6, p 494] is undefined on nested perform statements:

> Cobol 2002 Standard
>
> The results of executing the following sequence of PERFORM statements are undefined and no exception condition is set to exist when the sequence is executed:
>
> - a PERFORM statement is executed and has not yet terminated, then
> - within the range of that PERFORM statement another PERFORM statement is executed, then
> - the execution of the second PERFORM statement passes through the exit of the first PERFORM statement.
>
> NOTE Because this is undefined, the user should avoid such an execution sequence. On some implementations it causes stack overflows, on some it causes returns to unlikely places, and on others other actions can occur. Therefore, the results are unpredictable and are unlikely to be portable.

So, the behaviour of multiple active overlapping perform statements is undefined by the latest Cobol standard, and it does not forbid the use of recursive perform statements. We consulted a few reference manuals from several Cobol dialects on these issues. We summarise our findings here, illustrated by text snippets from the manuals.

- Micro Focus Cobol [152]:

  ".. an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement should not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements should not have a common exit.

  These restrictions are not enforced. PERFORM statements can be freely nested, and recursion (a PERFORM statement performing a procedure containing it) is allowed. Only the exit point of the innermost PERFORM statement currently being executed is recognized. These rules can be changed by use of the PERFORM-TYPE Compiler directive."

- IBM Cobol [100, 101]:

  "As an IBM extension, two or more active PERFORM statements can have a common exit."

  "A PERFORM statement must not cause itself to be executed. A recursive PERFORM statement can cause unpredictable results."

- Compaq Cobol [59] (formerly DEC Cobol) and DEC Cobol [72]:

  "Two or more active PERFORM statements cannot have a common exit.

  Use the check compiler option with the perform keyword to verify at run time that there is no recursive activation of a PERFORM."

- Acucorp Cobol [3]:

  "When this (compiler) switch is used, the PERFORM verb is modified so that return addresses are stored on a stack. Only the most recent PERFORM statement has an active return address. When this option is used, a paragraph under the control of a PERFORM statement may (directly or indirectly) PERFORM itself. See the configuration option PERFORM-STACK."

- PERCobol [134]:

  "PERCobol supports a return-stack implementation of PERFORM."

- Fujitsu Cobol [88]:

  "PERFORM statements within the range of containing statements cannot have a common exit. In this compiler, however, one or more PERFORM statements within the range of containing statements can have the common exit."

- Siemens Nixdorf Cobol [177]:

  ".. two or more active PERFORM statements must not have a common exit."

- RM Cobol [140]:

  ".. two or more active PERFORM statements may not have a common exit."

These reference manuals have no uniform semantics of a nested perform statement, and its behaviour can often be changed by setting compiler switches. To examine the behaviour of perform statements in practice, we compiled and ran three small test programs using different compilers and examined the output. The three test programs are shown in Figure 3.2, Figure 3.3 and Figure 3.4. Display statements are used to track the control-flow when a program runs. We briefly explain the programs here.

- **Program P1** This program has a nested overlapping perform statement. In `LABEL1`, a perform statement performs `LABEL2` through `LABEL3`. Then in `LABEL2`, a second perform statement references `LABEL3` through `LABEL4`, thereby passing control through the exit of the first perform statement.
- **Program P2** In this program, `LABEL3` is performed. The first time this paragraph is entered, variable `A` has value 1 and thus a goto statement jumps to `LABEL2` before the end is reached. After `LABEL2`, the control-flow can fall through to `LABEL3`. The second time this paragraph is entered, variable `A` has value 0 and thus the end of the paragraph is reached. When the perform statement in `LABEL1` terminates, the program displays `'END'`.
- **Program P3** This program has a recursive perform statement. In `LABEL1`, variable `A` is initialised with value 1 and `LABEL2` is performed. In `LABEL2`, the value of `A` is displayed. Then, if the value of `A` is less than 3, it is incremented and a recursive perform of `LABEL2` is made. If `A` is greater than or equal to 3, the program prints `'END'`.

We compiled these three programs with a number of different compilers on different platforms. We used AcuCobol [2], Compaq Cobol [58], Fujitsu Cobol [87], IBM Cobol [98], Micro Focus Cobol [151], and TinyCobol [85] (an open source Cobol compiler), the platforms were Windows, OpenVMS, OS/400, Unix and Linux. The compilers were initiated with no additional directives, except for the Micro Focus compiler, which we used also with the directive `PERFORM-TYPE=OSVS`. This directive provides compatibility with the mainframe behaviour of OS/VS Cobol. The directive `PERFORM-TYPE=RM` provides compatibility with the behaviour of RM Cobol and the directive `STICKY-PERFORM` also influences the behaviour; we have not tried these options in the experiment. We mention that none of the compilers issued any warning during the compilation of the test programs.

The output of the three test programs is shown in Table 3.1. The output of Program P1, with the nested perform statements, differs between compilers. In AcuCobol, Micro Focus Cobol and TinyCobol, the perform statements do not interfere with each other. First, `LABEL2` is performed, which performs `LABEL3` through `LABEL4`. Then, after the perform statement in `LABEL2` terminates, `LABEL3` is executed once more before the program ends. In Compaq Cobol, Fujitsu Cobol, IBM Cobol, and Micro Focus Cobol with the OS/VS directive, the nested perform statements do interfere. The first perform statement modifies the continuation point of `LABEL3`, and as soon as control-flow reaches that point, control-flow is transferred to the stop run statement after the first perform statement; `LABEL4` is not reached.

The output of Program P2 also illustrates the use of the perform continuation point. All compilers except TinyCobol produce similar results for this program. The perform

```
LABEL1.
    DISPLAY '1'
    PERFORM LABEL2 THRU LABEL3
    STOP RUN.
LABEL2.
    DISPLAY '2'
    PERFORM LABEL3 THRU LABEL4.
LABEL3.
    DISPLAY '3'.
LABEL4.
    DISPLAY '4'.
```

Figure 3.2: Program P1, with nested overlapping perform statements.

```
LABEL1.
    DISPLAY '1'
    MOVE 1 TO A
    PERFORM LABEL3
    DISPLAY 'END'
    STOP RUN.
LABEL2.
    DISPLAY '2'.
LABEL3.
    DISPLAY '3'
    IF A = 1
      MOVE 0 TO A
      GO TO LABEL2
    END-IF.
```

Figure 3.3: Program P2, where a goto statement jumps out of a performed paragraph.

```
LABEL1.
    MOVE 1 TO A
    PERFORM LABEL2
    STOP RUN.
LABEL2.
    DISPLAY A
    IF A < 3
      ADD 1 TO A
      PERFORM LABEL2
    END-IF
    DISPLAY 'END'.
```

Figure 3.4: Program P3, with a recursive perform statement.

Table 3.1: **Output of the three test programs.**

	AcuCobol[a]	Compaq[b]	Fujitsu[c]	IBM[d]	Micro Focus[e]	Micro Focus[f]	Tiny Cobol[g]
P1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2
	3	3	3	3	3	3	3
	4				4		4
	3				3		3
P2	1	1	1	1	1	1	1
	3	3	3	3	3	3	3
	2	2	2	2	2	2	2
	3	3	3	3	3	3	END
	END	END	END	END	END	END	
P3	1	1	1	1	1	1	1
	2	2	2	2	2	2	2
	3	3	3	3	3	3	3
	END	END	END..	END..	END	END..	END
	END	END	loop	loop	END	loop	END
	END				END		END

[a] AcuCobol Development Suite 4.3 for Windows & Development System 6.2 for Linux
[b] Compaq Cobol V2.7 for OpenVMS
[c] Fujitsu NetCobol V7.0 & Cobol 85 V3.0 for Windows
[d] IBM Cobol/400 & ILE Cobol/400 for OS/400
[e] Micro Focus ObjectCobol 4.1 for Unix
[f] Micro Focus ObjectCobol 4.1 for Unix, with compiler directive `PERFORM-TYPE=OSVS`
[g] TinyCobol V0.61 for Linux

statement sets the continuation point of `LABEL3` to `DISPLAY 'END'` in `LABEL1`. As long as the end of that paragraph is not reached, the continuation point remains active. If the program flow reaches that point at a later moment, the flow is transferred to `DISPLAY 'END'` in `LABEL1`. So, the continuation point is preserved even though the control-flow has left the performed paragraph. This can make comprehension and modification of programs difficult.

The output of Program P3 shows that Fujitsu Cobol and IBM Cobol do not support recursion by default, as well as Micro Focus Cobol with the OS/VS directive. The programs printed 'END' continuously and did not terminate. Compilers supporting recursion yielded programs that printed 'END' three times. The Compaq Cobol compiler also does not support recursion, but the test program terminated after it printed 'END' two times. We briefly discuss the IBM and Compaq implementations of the perform statement in more detail.

The implementation of the IBM Cobol compiler is discussed in [81], stating that each perform statement can save exactly one pending continuation point, i.e. a continuation that was set by an earlier perform statement and is still active. This works as follows: when a perform statement is executed, the pending continuation point of the performed paragraph is stored in an area specific for that perform statement, and a new continuation point of the performed paragraph is set to the current perform statement's successor. When control is returned after the current perform statement terminates, the stored pending continuation point is restored and control continues with the perform statement's successor.

But when a perform statement is executed for a second time before its first execution has terminated, the pending continuation point is overwritten before it was restored. So, to see what happened in our test Program P3 we reconstruct its execution. The `PERFORM LABEL2` statement in `LABEL1` set the continuation of `LABEL2` to the stop run statement in `LABEL1`. Then, the first time `PERFORM LABEL2` in `LABEL2` was executed, the pending continuation of `LABEL2` to the stop run statement was saved and the new continuation of `LABEL2` was set to `DISPLAY 'END'`. However, when `PERFORM LABEL2` in `LABEL2` was executed for the second time, the pending continuation point to the stop run statement was overwritten before it was restored. Then, each time the end of `LABEL2` was reached, its continuation was restored from `PERFORM LABEL2` in `LABEL2`, but that was always the `DISPLAY 'END'` statement; hence the program did not terminate.

The Compaq Cobol compiler also does not support recursion, but uses a different model than the other compilers. It produced a program which printed 'END' only two times before it terminated, whereas the other compilers either did not terminate or printed it three times since `LABEL2` was recursively executed three times. We examined the output of the debugger and it turned out that recursive calls can be made, but a problem occurs as soon as the end of `LABEL2` is reached for the second time. At the end of `LABEL2`, first the continuation to `LABEL1` was stored, and then overwritten two times by continuations to the `DISPLAY 'END'` statement, before the end of `LABEL2` paragraph was reached for the first time. When the end was finally reached, there was one continuation, referring to `DISPLAY 'END'`. After the execution of that statement, it seemed that another continuation was expected at the end of `LABEL2` but not found, and the program was terminated. The Compaq compiler can detect recursive calls by using a compiler option, but that option was not enabled and thus the problem was skipped without a warning. We assume that when using the Compaq compiler, precisely one continuation can be stored for each paragraph, whereas with the IBM compiler precisely one *pending* continuation can be stored for each perform statement. This means that in Compaq Cobol, one cannot have two or more active perform statements with the same exit, whereas in IBM Cobol one cannot have two or more simultaneous activations of the same perform statement. This is in line with the manuals we consulted. The subtle difference was exposed by the recursion in the test program, but can also be demonstrated by employing a program that has two active performs with a common exit.

By using three small test programs, we demonstrated that there are at least three different implementations to store the continuations for perform statements, and these implementations cause different behaviour of the test programs.

3.2.3 Hazardous control-flow: Cobol minefields

The test programs in the previous section illustrate that seemingly simple procedure calls in Cobol can cause non-intuitive behaviour and portability problems. We demonstrated that there exist different implementations to store the continuations of performed paragraphs (or sections).

Some of our findings are also discussed in [81], where a Cobol mine is considered to be an active continuation of a perform statement that is skipped due to a goto statement. The mine can be tripped unexpectedly at a later moment. However, whether a mine can

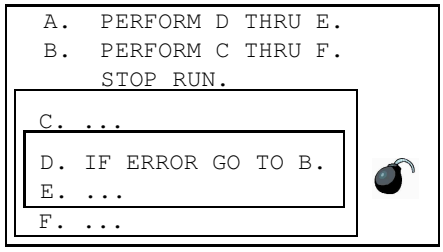

Figure 3.5: This code example was adopted from [81]. In case of error in paragraph D, the first perform statement leaves a mine at the end of paragraph E, which the subsequent perform can trip. However, this can only happen in IBM-like implementations of the perform statement.

be tripped may depend on the used Cobol compiler. For instance, if the mine is reached at a later moment during the execution of a second perform statement, the mine does not have to be active since some compilers have only one active continuation at a time. In the example program in Figure 3.5, which we adopted from [81], a mine can be activated and subsequently tripped. The first perform statement sets the continuation of paragraph E to B. If ERROR evaluates to true, control-flow is transferred to B by a goto statement but the continuation to B at the end of E remains active. Then, in B, paragraphs C through F are performed. Subsequently, if the error in D does not occur for the second time, the mine at the end of E can be tripped; however, this depends on the compiler. If the code is compiled with a compiler with only one active continuation at a time (e.g. AcuCobol, Micro Focus, Tiny Cobol), the mine cannot be tripped. If the code is compiled with a compiler with multiple active continuations, the mine *can* be tripped (e.g. Compaq, IBM, Fujitsu). On the other hand, if the mine is reached by means of a goto statement or fallthrough logic, (e.g. in B there is a goto statement to C instead of the perform statement) it is tripped regardless of the used compiler; we illustrated such behaviour with test program P2 in Figure 3.3.

In this chapter, we consider a number of potential hazardous programming practices as Cobol mines, and code containing these structures is regarded as a Cobol minefield. A mine involves a range of performed labels combined with one of the three main ways to transfer control-flow in Cobol: perform, goto and fallthrough. Potential mines are performed ranges that can be entered or exited by goto or fallthrough logic, or performed ranges that overlap with other performed ranges. Hence, we identified three main types of mines, classified as a *perform mine*, a *goto mine*, or a *fallthrough mine*. For the goto and fallthrough mines there are two distinct subtypes, indicated by *Into* (e.g. a goto jumps into a range) and *Out* (e.g. a goto jumps out of a range). The description and definition of mines is shown below. Each mine involves a performed range R combined with one or more labels denoted by $L_1, \ldots, L_n$, with a single label denoted by L, or with a statement denoted by S.

- *perform mine*

  one or more labels $L_1, \ldots, L_n$ which appear in two dissimilar performed ranges R_1 and R_2 (overlapping ranges, multiple entry and exit points)
  $L_1, \ldots, L_n : L_1, \ldots, L_n \in R_1 \wedge L_1, \ldots, L_n \in R_2 \wedge R_1 \neq R_2$

- *goto mine*

  Into: a goto statement S that is not contained in a performed range R, which references a label L contained in R (jump into a range, multiple entry points)
  $S : S = gotoL \wedge S \notin R \wedge L \in R$

  Out: a goto statement S contained in a performed range R, which references a label L that is not contained in R (jump out of a range, multiple exit points)
  $S : S = gotoL \wedge S \in R \wedge L \notin R$

- *fallthrough mine*

  Into: the first label L in a range of performed labels R which can be entered by fallthrough logic from its syntactic predecessor (multiple entry types)
  $L : L = \mathit{first}(R) \wedge \mathit{fallthrough}(predecessor(L), L)$

  Out: the last label L in a range of performed labels R, which can be exited by fallthrough logic to its syntactic successor (multiple exit types)
  $L : L = \mathit{last}(R) \wedge \mathit{fallthrough}(L, successor(L))$

Some paragraph labels may be classified as several types of mines, which can be an indication of very intricate control-flow. For example, a label can be classified as both a perform mine and a fallthrough mine. Note that the above definitions for fallthrough mines are not complete, but they are sufficient to describe mines. The predicates $successor()$ and $predecessor()$ can be determined with little effort. The predicate $fallthrough()$ requires more effort, since it cannot be seen from a paragraph itself. In fact, it is required to track the control-flow paths from the beginning of a program to the specified paragraphs. Therefore, the use of automatic tools to carry out large-scale minefield detection is inevitable.

The mines we defined can cause unexpected behaviour and portability problems, and complicate program understanding and modification; code with mines leads to multiple entry and exit points in a segment of code. Although the use of multiple and single entry and exit points in programming can be subject to debate, in Cobol one has to be especially careful with different types of entry points because these influence the way the control continues. One also has to be careful with multiple exit points because of the peculiar preservation of continuation points. Moreover, when we read the fine prints of the Cobol standard [6, p 491] we found for a perform statement referencing 'procedure-name-1' through 'procedure-name-2' that:"*the flow of execution should eventually pass to the end of procedure-name-2*". Hence, if a program has a path that jumps out of a performed range and does not reach the end of its last paragraph or section before the program terminates, it is not in line with the Cobol standard.

Hence, we believe it is wise to disallow mines in a coding standard because mines violate best practices and can be harmful to the operation and maintenance of a system. By

detecting mines in advance, one can prevent a system from high costs for error correction and system failure, and reduce risks during updates and migrations.

3.3 The Minefield project: Cobol minefield detection

In the Minefield project, we implemented a mine detector to identify Cobol minefields and applied it in a case study on minefield detection. The case study contained five business-critical Cobol systems from different companies, covering about 830 thousand lines of code in total. We discuss our tools, some implementation details, and then the case study.

3.3.1 Tools and implementation

Mine detection and control-flow analysis In order to detect Cobol mines, control-flow analysis is required. We implemented a static control-flow analysis tool for Cobol. Using the output of the analysis tool, we implemented a mine detector. The control-flow analysis tool visits and collects all possible control-flow paths in a program. For proper control-flow analysis, one needs to have precise semantics of the control statements. Since the semantics for the perform statement differs among implementations, we chose a semantics: when several perform statements are active at the same time, only the continuation of the innermost perform statement is recognised. If the continuation of the innermost perform is reached, control is returned and the continuation of the outer perform becomes active again. So, our semantics implement a stack mechanism, resembling the behaviour we encountered with the Micro Focus, Acucorp, and Tiny Cobol compilers. We believe that this semantics is easier to understand and more in line with the concepts of structured programming than a semantics with multiple active continuations, which we found in the Compaq, Fujitsu and IBM Cobol compilers.

Different semantics can yield different reachable points in a program. In some rare cases, code can be reachable when one continuation is active, whereas it is unreachable when multiple continuations are active, and vice versa. This means that we can miss a mine with our mine detector because the mine occurs in code that is considered not to be reachable, i.e. *dead code*. We illustrate this with a code sample in Figure 3.6. In the code, if only a single continuation can be active at a time, the code in `LABEL2` is not reachable. If multiple continuations can be active at the same time, the code in `LABEL6` is not reachable. The code sample shows that we may miss mines that are located in code that is unreachable according to the semantics we use for the control-flow analysis. The differences in reachable code under different semantics are caused by the existence of mines (e.g. the code in Figure 3.6 has overlapping perform ranges and a jump out of a perform range). Of course, that is one of the goals of minefield detection: the discovery of code that behaves differently when it is compiled with different compilers. Hence, in the cases where we miss a mine due to the above circumstances, there is at least one other mine in the program that is responsible, and that mine is detected.

Visualisation tool We implemented a visualisation tool that translates the output of the control-flow analysis to graphs in Dot format. Dot is part of the Graphviz package [10]

```
LABEL1.  PERFORM LABEL4 THRU LABEL5.
LABEL2.  DISPLAY 'THIS CAN BE DEAD CODE'.
LABEL3.  STOP RUN.
LABEL4.  PERFORM LABEL5 THRU LABEL6.
LABEL5.  ...
LABEL6.  DISPLAY 'THIS CAN BE DEAD CODE'.
         GO TO LABEL3.
```

Figure 3.6: Different perform statement semantics can cause different program points to be reachable.

and can be used to draw directed graphs. A directed graph is convenient for visualisation of the control-flow in a Cobol program, as it is usually directed in a top-down, hierarchical way. Our visualisation tool generates control-flow graphs that provide a comprehensive overview of the control-flow among paragraphs and sections in a program. This way, minefields can be visualised. In Figure 3.7, an example program with its generated graph is shown. The blocks represent sections and the ellipses represent paragraphs. The dashed (green) arrow represents a perform statement and thick (red) arrows goto statements. The number next to an arrow indicates in which order the control statements appear in a paragraph; in this case there are two goto statements in the paragraph `SUBPAR`, the first one jumps to the exit paragraph and the second one is a local loop. The thinner (black) arrows represent fallthrough. Since `PERFORM SUBROUTINE` in `MAINPAR` can return control at the end of `SUBROUTINE`, there exists a fallthrough arrow from `MAINPAR` to `ENDPAR`.

Implementation details To carry out source code analyses fast and reliable, one can use automatic tools [40, 173, 199]. In our case, we used the ASF+SDF Meta-Environment [45, 48, 112] to implement the control-flow analysis tool and the mine detector, as well as the tool to visualise graphs. We refer to Chapter 2 for detailed information on the use of ASF+SDF and the ASF+SDF Meta-Environment for source code analysis.

In the case study, we used a Cobol grammar in SDF for parsing the Cobol programs. The grammar was derived from the online IBM VS Cobol II grammar [129, 130, 131] and modified [116] to be able to parse the Cobol dialects in the case study. To aid the adaptation of the grammar, we used the Grammar Deployment Kit (GDK) [122]. We employed a preprocessor for the Cobol programs that was adopted from [41], and the architecture we used for carrying out automated analysis and transformation of the Cobol programs is described in [44, 173, 199]. More details on preprocessing, parsing, and the infrastructure can also be read in Chapter 2. Furthermore, we constructed a Dot grammar in SDF using the grammar from the Dot manual [89, p 34]. The Dot grammar was used for our visualisation tool. The control-flow analysis, the mine detector, and the visualisation tool were implemented using ASF rewrite rules. We briefly describe the operation of the tools, using the program in Figure 3.7 as a running example.

The control-flow analysis explores all possible paths in a program, starting at the first statement in the procedure division. Only control statements are considered (i.e. goto statements, perform statements and statements terminating the execution of the program,

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
PROCEDURE DIVISION.
MAIN SECTION.
MAINPAR.
    PERFORM SUBROUTINE.
ENDPAR.
    STOP RUN.
SUBROUTINE SECTION.
SUBPAR.
    IF FLAG
        GO TO SUB-EXIT
    ELSE
        DISPLAY 'FLAG NOT SET'
    END-IF
    GO TO SUBPAR.
SUB-EXIT.
    EXIT.
```

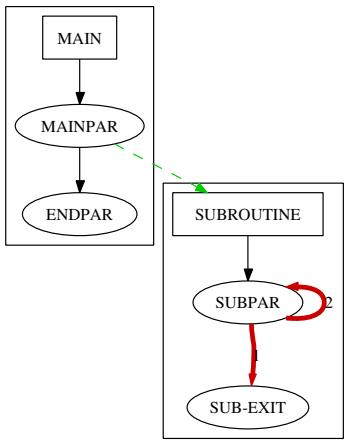

Figure 3.7: Example program and its generated control-flow graph. The dashed green arrow represents the perform statement, the thick red arrow represents the goto statement, the thin black arrows represent fallthrough logic, and the numbers represent the order of appearance in the code.

and branching statements containing a control statement); all other statements are ignored. During the traversal of the program, each visited statement is stored in a list, representing the current path. In addition, a list of active perform statements is maintained. At the end of a paragraph, the list is queried to see if the paragraph is the current active continuation. If it is, control continues with the statement after the active perform statement. Otherwise, control falls through to the next paragraph and continues with its first statement. If there is no next paragraph (i.e. the end of the program is reached), the traversing stops and the current path is returned. A branching statement splits the current path, resulting in two or more possible paths. If a loop is encountered, the traversing stops and the path so far is returned. A loop occurs if a statement has already been visited within the scope of the current active continuation (i.e. a goto transfers control to a previously visited statement), or if a perform statement is already in the list of active perform statements (i.e. a recursive perform). The example program from Figure 3.7 has the following two possible paths (the scope of an active continuation is indicated by the curly brackets):

1. `PERFORM SUBROUTINE { BRANCH GO TO SUB-EXIT } STOP RUN`
2. `PERFORM SUBROUTINE { BRANCH GO TO SUB-PAR BRANCH }`

Each path starts at the first statement in the program, which is `PERFORM SUBROUTINE`. When the if statement is encountered (indicated in the path by `BRANCH`), the first path takes the if-branch containing the `GO TO SUB-EXIT` statement. At the end of `SUB-EXIT` (the exit statement is ignored), the active continuation of `PERFORM SUBROUTINE` returns control, and the next executable control statement is the `STOP RUN` statement, which ends program execution and thus the analysis of the first path. The

second path takes the else-branch and encounters the `GO TO SUBPAR` statement (the display statement is ignored). Control is again directed to `SUBPAR`, and the if statement is visited for the second time within the scope of the same active continuation. That indicates a loop, and the analysis of the second path ends and the path that has been constructed so far is returned. The algorithm always terminates because the number of statements in a program is finite. Note that our control-flow analysis must use a little more information than what is shown in the two example paths. In order to detect a loop, each statement is qualified by a unique identifier. We left this out in order to provide a concise example.

The retrieved control-flow data is used by the visualisation tool to draw a control-flow graph, depicting the dependencies between the paragraphs. For example, according to the first path, there is fallthrough from the `PERFORM SUBROUTINE` statement in `MAINPAR` to the `STOP RUN` statement in `ENDPAR`. We implemented a transformation to construct a graph in Dot format from the retrieved information. Each edge is represented by a left-hand side and a right-hand side, and a number of attributes. Attributes are used to manipulate edges; for instance, attributes can specify styles, colors, labels and weights (the weight attribute specifies the cost for stretching an edge; this can be used to influence the positioning of the nodes). An edge that represents a goto statement can be specified in a single Dot statement. Similarly, edges for fallthrough and perform statements can be drawn, as well as the structures for the sections and paragraphs. Using Dot, we were able to specify graphs concisely; the entire Dot specification for the example program from Figure 3.7, which is presented in Figure 3.8, has about 20 lines of code.

The main goal of the Minefield project is of course to detect minefields. The mine detector uses the control-flow data to detect mines in a program. In order to detect perform mines, we gather all perform ranges (i.e. a list of paragraph labels) from the control-flow data and search for overlapping ranges. According to our definition, this means that we detect paragraph labels that occur in dissimilar ranges. In the example program, there is only one perform range, containing labels `SUBPAR` and `SUB-EXIT`. To detect goto mines, we proceed in a similar way. For the goto Into mines, for each perform range, we detect goto statements that occur outside the range, which refer to a label in the range. For goto Out mines, for each perform range, we search goto statements that occur in the range, which refer to a label outside the range. For the fallthrough mines, we search for the first label in a perform range that can also be reached by fallthrough from its predecessing label, and the last label in a perform range that has an outgoing fallthrough edge to its successor. The example program from Figure 3.7, which we used as a running example so far, has no mines. In the next section, we present several real programs containing harmful mines.

3.3.2 Case study

We applied our minefield detection to five industrial Cobol systems that are in production and being maintained by different companies. In total, the systems covered approximately 830,000 lines of Cobol code in nearly 1,900 programs. We show some statistics in Table 3.2 to illustrate the diversity of the systems in terms of lines of Cobol code, number of programs, sections, paragraphs and control statements perform and goto. Consider, for instance, the amount of goto statements and the number of lines of code in System 3 com-

```
digraph "EXAMPLE" {

 subgraph "cluster_MAIN" {
   "MAIN" [shape=box] ;
   "MAINPAR";
   "ENDPAR";
 };

 subgraph "cluster_SUBROUTINE" {
   "SUBROUTINE" [shape=box];
   "SUBPAR";
   "SUB-EXIT";
 };

 "MAIN"       -> "MAINPAR"    [weight=100];
 "MAINPAR"    -> "ENDPAR"     [weight=100];
 "MAINPAR"    -> "SUBROUTINE" [color=green,style=dashed];

 "SUBROUTINE" -> "SUBPAR"     [weight=100];
 "SUBPAR"     -> "SUB-EXIT"   [color=red,style=bold,label="1"];
 "SUBPAR"     -> "SUBPAR"     [color=red,style=bold,label="2"];
}
```

Figure 3.8: Dot code for drawing the graph from Figure 3.7.

pared to System 4. We started with System 1, a medium sized system of nearly 80,000 lines of code, and evaluated the results with a system expert of that system. After that, we deployed our minefield detector on the rest of the systems.

In Table 3.3, the results of the minefield detection are summarised. An interesting finding is that in System 4, which is almost 200,000 lines of code, we found only four mines. We assume that this has to do with the low number of goto statements in that system, which can be an indication that the chances on mines are lower in a system with less goto statements. Another characteristic that appears from the data in the tables is that the detected mines are not equally distributed over a system. In fact, a relative small number of programs in a system are responsible for mines. This characteristic is in line with findings of others [25, 79, 157]: a small number of modules in a system are responsible for the majority of the errors. Such a characteristic is known as a Pareto distribution, or the 80:20 rule: 80 percent of consequences originate from 20 percent of the causes. In our case, if we sum up all program of the systems, 93 of the 1,886 programs are responsible for all mines, which corresponds to 5% of the programs. It is interesting to investigate whether there is a correlation between the programs responsible for mines and the original programmers, their background, education, and other aspects that may influence programming styles. However, it is usually difficult to receive feedback on programs and changes that were initiated a long time ago. Hence, we were not able to validate all of the reported minefields with the owners of the systems.

Table 3.2: Statistics of the systems in the case study.

System	loc Cobol	programs	sections	paragraphs	performs	gotos
1	78,253	92	1,406	2,892	2,308	872
2	17,168	31	434	940	825	105
3	79,867	85	1,286	3,723	2,040	2,633
4	198,384	833	3,089	6,311	5,079	12
5	457,310	845	9,979	21,124	17,513	7,724
total	830,982	1,886	16,194	34,990	27,765	11,346

Table 3.3: Results of the minefield detection

System	reported	total	perform	goto		fallthrough	
	programs	mines	mines	Into	Out	Into	Out
1	14	491	-	30	35	213	213
2	9	19	18	-	-	1	-
3	6	43	8	-	1	18	16
4	3	4	2	-	-	2	-
5	61	413	153	26	42	96	96
total	93	970	181	56	78	330	325

To verify whether our mine detector correctly identified minefields (i.e. reported minefields actually occurred in a program), we analysed the reported minefields. We looked at the reported paragraph labels, and, in case of goto mines, the accompanying goto statements. We did not inspect each reported program, as several of the reported paragraph labels showed similar patterns. For example, by studying the names of labels, we were able to determine that certain overlapping perform ranges involved a section that performs one of its inner paragraphs. We investigated some of the programs with such patterns, and at a certain point we were confident that similar patterns represented the same kind of minefield. We did investigate each program with an abnormal pattern, which turned out to be the most intricate minefields. Several of these minefields will be discussed in this section.

We now discuss our findings with System 1 in more detail. After that, we elaborate on the results of the other four systems.

System 1 In the first system, we detected 491 mines in 14 programs. Most mines were fallthrough mines: a performed range that can also be entered or exited by fallthrough logic. In all detected cases, each range that can be entered by fallthrough was preceded by a range that can be exited by fallthrough. Therefore, the number of fallthrough Into mines is equivalent to the number of fallthrough Out mines. The cause of all the fallthrough mines was the presence of goto mines: a jump into or out of a range. We illustrate this with the control-flow graph of one of the reported programs in Figure 3.9, which also illustrates the problems associated with these mines. The program reads records from a file and updates the database if necessary. Section `B11_INIT` uses `X01_READ_FILE` to read the first record and establishes a connection with the database.

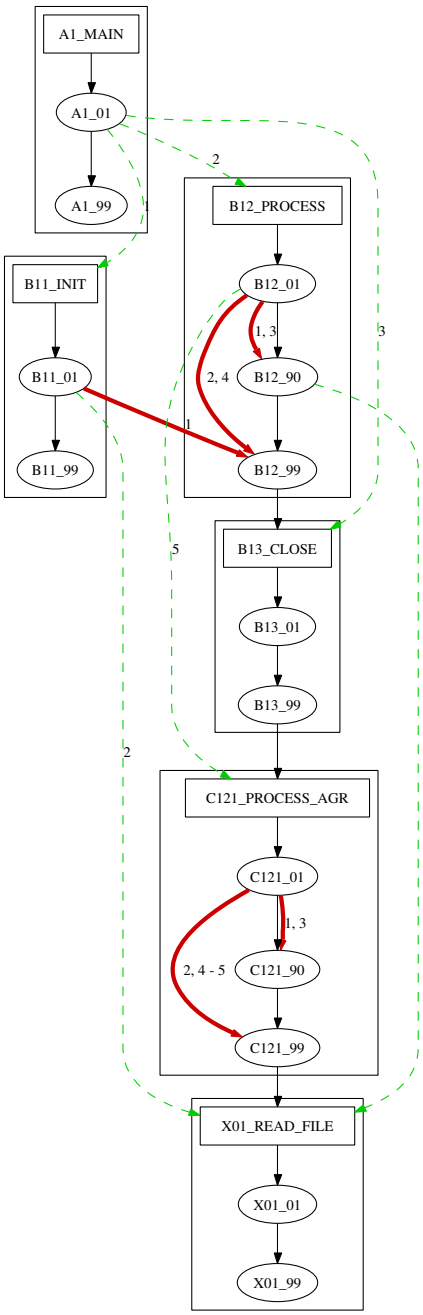

Figure 3.9: A Cobol minefield in System 1. The erroneous goto statement from section `B11_INIT` to section `B12_PROCESS` violates the corporate coding standard and causes unintented fallthrough between all subsequent sections.

Then, B12_PROCESS is executed until there are no more records in the file or an error occurs; C121_PROCESS_AGR is used to determine if the database must be updated. If all records are processed, B13_CLOSE closes the file, prints the number of read records, and commits the changes to the database. If an error occurs, B13_CLOSE performs a rollback operation and another program is called. That program handles and logs the error, and, when it terminates, control is returned to B13_CLOSE. After B13_CLOSE terminates, the program is ended by a stop run statement in A1_MAIN. However, in this program, there is a goto statement from paragraph B11_01 to paragraph B12_99 in section B12_PROCESS. This jump creates fallthrough between all subsequent sections until the end of the program. In total, this program contains eight mines: two goto mines and six fallthrough mines.

An interesting finding was the following: since a jump out of a range will often result in a jump into another range, one would expect that the number of goto Into mines (jump into of a range) is equivalent to the number of goto Out mines (jump out of a range). However, we detected 35 goto Out mines but only 30 goto Into mines. A closer examination of the code revealed that some of the reported goto mines jumped to a label inside a section that was not reachable otherwise. Hence, if the jump was programmed by mistake, it accidentally revived some dead code. This results in *zombie code*: code that used to be dead but which has been revived.

As we suspected that the mines in this system were not implemented intentionally, they created possible execution paths that were never intended to exist and may result in severe errors. For instance, incorrect data can be stored or displayed, or the program terminates abnormally. Besides that it can be very difficult to trace the error, the system may be down for a while after such errors occur. So, we wanted to know how many of the detected mines were implemented intentionally and how many where implemented unintentionally. However, that may be difficult to determine for sure without proper knowledge of the system. All we saw from the code was that there were jumps out of a performed section, which can be intentional as well as unintentional. Therefore, we consulted one of the system experts of System 1, and asked him his opinion about the 14 reported programs. It turned out that jumps between sections were not allowed in the corporate coding standard, and the system expert was confident that all cases were programming errors. Often, this was due to a sloppy copy-paste of code. So, in System 1, all mines were implemented unintentionally and thus the fallthrough execution paths and zombie code were also created unintentional.

To have a better sense of the degree to which a minefield causes errors, we tried to assess the severity of the mine from Figure 3.9 in more detail. In B11_INIT, the input file is opened, and if something is wrong with the file, an error routine is called and a (global) error switch is set. After this switch is set, the erroneous goto statement is executed and transfers control to B12_99 unintentionally. This paragraph has only an exit statement, and control continues with B13_CLOSE because the continuation of B12_PROCESS is not active. In B13_CLOSE, the input file is not closed because it was not properly opened, and no commit to the database is done because no connection was established. The number of read records is displayed, which is zero, and, because the error switch was set, another program is called to handle the error. After that, control falls through to C121_PROCESS_AGR. In that section, it is attempted to retrieve data from

the database but that fails because no connection exists. This results in an error that is handled by the error routine for the database connection. The error switch, which was already set, is set again, and after the error routine terminates, control is transferred to C121_99, which contains just an exit statement. At the end of C121_PROCESS_AGR, there is no active continuation and control falls through to X01_READ_FILE. In that section, a record is read from the input file, and after that the end of the program is reached, which terminates the execution.

To summarise, the program in Figure 3.9 has a valid execution path that is clearly erroneous. If something is wrong with the input file in this program, control-flow proceeds in an unintended way through the program. Along the way, several other programs are called and error logs are created that are not relevant to the actual error. This can obstruct proper maintenance and operation of the system severely. The error has either shown up already or not yet. We suspect that, in case this error has shown up already, the maintainers have been confused by the output and the error logs. At first sight, it appeared that something is wrong with the database. After several error logs are examined and some tests are ran, it turns out that there was an error with the input file. The maintainer fixes the problem with the file, reruns the program, and the error has disappeared. In either case, the mines remain dormant in the program until they show up, threatening the operation of the system and causing unnecessary maintenance expenses.

The errors associated with the minefields in this system either have not yet occurred or a system expert knows what to do when these errors arise. The example illustrated that mines can cause code to be executed in an unintended order, resulting in unexpected output that is difficult to debug. A dormant error can also be activated when the program is modified. Errors, like the ones we found in this system, can be fixed in advance if the erroneous labels are corrected. We illustrate this in Figure 3.10, which depicts the graph of the repaired program from Figure 3.9. The jump from B11_01 to B12_99 was replaced by a jump from B11_01 to B11_99. This way, the fallthrough mines in the subsequent sections are also removed. Hence, if the erroneous jump is repaired in this program, all mines are cleared. Nevertheless, a mine should always be cleared with care. For example, there can already be a work-around for a mine, and by removing the mine we can actually introduce a new error.

We delivered a report to the company with the detected mines in the system and the control-flow graphs. Some of the system experts were especially interested in the fallthrough analysis between sections, because they had been confused by this behaviour during maintenance. This indicated that they had already suffered from minefields. After we reported the mines in System 1 and evaluated the results, we also analysed four other systems. For each system, we discuss the results separately, illustrated with some appealing examples.

System 2 System 2 is a small system of only 31 programs, covering about 17,000 lines of code. In total, 19 mines were detected in 9 programs; 18 were classified as perform mines and one as a fallthrough Into mine (fallthrough into a performed range). The fallthrough mine appeared in a small program of only two sections, which is shown in Figure 3.11. The first section performs the second one, but after that the control-flow executes the second one again by fallthrough; this was an error. The first section should have been

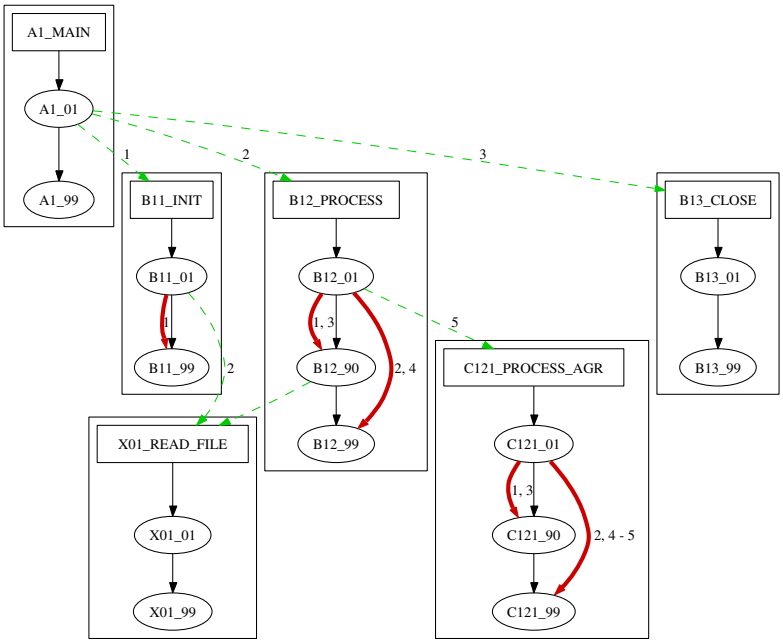

Figure 3.10: The repaired System 1 program from Figure 3.9.

```
PROCEDURE DIVISION.
MAIN SECTION.
MN_00.
    PERFORM A01_PROCESS.
MN_99.
    EXIT.

A01_PROCESS SECTION.
A01_00.
    MOVE 5 TO CUST_TYPE.
    ...
A01_99.
    EXIT.
```

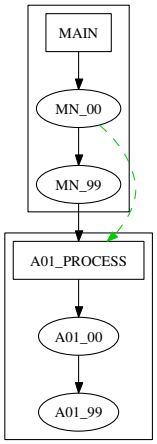

Figure 3.11: A Cobol fallthrough mine in System 2. The `A01_PROCESS` section is reachable by both a perform statement and by fallthrough logic.

```
     PERFORM C-DAT.
...
C-DAT SECTION.
...
C09. IF COUNT-BT > COUNT-1
        GO TO C99.
     ...
     IF P-CODE = ZERO
        ...
        GO TO C09.
     ...
C11. ...
     SEARCH TAB-LK
        AT END GO TO C99
        WHEN T-CODE = LK-1
             GO TO C13.
C13. ...
     IF AK-1 = ZERO
        GO TO C15.
     ...
C15. ...
C17. ...
     PERFORM C11 THRU C15
     ...
     GO TO C09.
...
C99.
     EXIT.
```

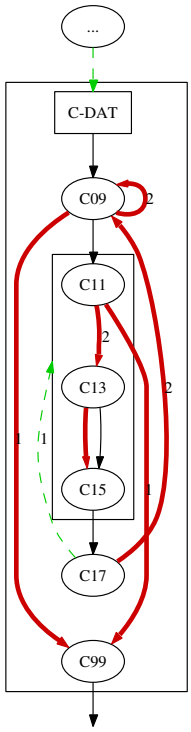

Figure 3.12: A Cobol minefield in System 3.

terminated with an exit program statement but instead there was just an exit statement (which has no effect on the execution of a program). The consequence is that the code in the second section is unintentionally executed twice instead of once.

The 18 overlapping perform ranges were caused in a performed section, which performs some of its own inner paragraphs. Although this is perfectly legal in Cobol, this way of programming may lead to errors. If an internal range of performed paragraphs must only be reached by perform statements, a goto statement is required to skip the range when reached by fallthrough inside the section. If the inner range can also be reached by fallthrough (e.g. under different input conditions), we have a fallthrough mine: the flow of control after the inner range is determined by the way the range was reached. This complicates comprehension of the code, and we found an example of such a situation in System 3. In addition, a performed range inside a performed section can pose portability problems, since there is more than one active perform range inside the same section; a programmer might be tempted to implement a jump from the inner range to the outer range since the inner range is directly contained within a section. We will show examples of such code later. On top of this, we read in one of the compiler manuals [153, p 28] that when an end of a perform range lies within another range, the compiler produces very inefficient code. So, although the overlapping perform ranges in this system are

programmed intentionally and cause no error at the moment, the coding style is prone to errors and can be inefficient.

We reported our findings to the company that maintained the system. The results were handed to the application manager of the system. The manager told us that she was eager to investigate our findings with the system experts, but that maintenance of the system was frozen at that moment, and thus no further preventive maintenance could be carried out at the time of writing this chapter.

System 3 In System 3, also a medium sized system of about 80,000 lines of code, 43 mines were reported. The perform mines and many of the fallthrough mines were caused by a performed section which performs some of its own paragraphs. Similar practice was also found in System 2, which we discussed above, but now the inner ranges are also reachable by fallthrough logic, and there were jumps from the inner range to the outer range. This means that the first paragraph of the inner range is classified as both a perform mine and a fallthrough mine, which results in very intricate code. We show a simplified code snippet from System 3 together with its visualisation in Figure 3.12. The code implements a loop in which a table is searched; the entire section was about 150 lines of code, containing 31 goto statements and 21 paragraphs. The actual searching is done in paragraph `C11`, but that paragraph can be reached in two ways: by direct fallthrough from paragraph `C09` and by a perform statement in paragraph `C17`. This results in several mines that complicate the code: an overlapping range (`C11 THRU C15`), a jump out of a range (in `C11 GO TO C99`), and a range that can be entered and exited by fallthrough (`C09` into `C11 THRU C15` to `C17`). In addition, this code has portability problems since it behaves differently on some compilers. On a compiler with a single active perform continuation, if `C11` is reached by the perform statement in `C17`, the continuation of the outer perform statement (`PERFORM C-DAT`) is not active. Then, if in `C11` the goto statement to `C99` is executed, control-flow falls through from `C99` to the next section in the program and control is not returned to `PERFORM C-DAT`; hence, there is an additional fallthrough mine at the end of `C99` which occurs when certain compilers or compiler flags are used.

It would be interesting to interview the original developer of this code, but he was no longer maintaining this system. Hence, we were unable to receive proper feedback on this particular minefield. We assume that the complexity of the program has evolved over time, causing a tangled structure. For a new programmer, the code is difficult to understand and modify, it is prone to errors, and it has portability problems. One could untangle the code by restructuring it; however, if this particular piece of code is untangled fully automatic, it will result in code that is still difficult to grasp. In addition, to clear this minefield automatically, an automatic tool has to be tailored to some semantics for a perform statement, because the code is not in line with the Cobol standard. Hence, a more appropriate solution to such minefields is to detect, record and report such code, and let a system expert restructure the code.

System 4 System 4 is a larger system with nearly 200,000 lines of code in 833 programs. A striking statistic is the low number of goto statements: only 12. The system was initiated in the early 90s, and apparently the use of goto statements was restricted during implementation and maintenance. The result is hardly any goto statements, but in return

there are some complex loops with deeply nested if statements and flag variables. Our mine detector reported four mines in this system: two perform mines, and two fallthrough Into mines. One of the fallthrough mines was similar to the one found in System 2, where a section can first execute a subsequent section by a perform statement and then executes the subsequent section again by fallthrough. In this case, the mine was less dangerous because the perform statement referencing the second section was located in an if statement and thus can only be executed conditionally. The second section contained an unconditional program terminating statement, ending the execution of the program; hence, the second section never returned control and thus cannot be executed two times in a row in a single program execution. Still, the program contained an error that can cause problems sooner or later.

The two perform mines and the other fallthrough mine in the system were caused by a programming error. We show a code sample in Figure 3.13 containing this error. In the code, it can be seen from the naming convention and section structure that the original intention was to create two subsequent sections; however, in the second section label `X11_CONT_CHANGED`, the keyword '`SECTION`' was omitted. The compiler now assumes the label `X11_CONT_CHANGED` is an empty paragraph in `X01_COMMIT_CHECK`. If the condition in `X01_00` is true, `PERFORM X11_CONT_CHANGED` executes no statements. When that statement terminates, control-flow continues to `X01_99` and then falls through to `X11_CONT_CHANGED`. Then the statements in `X11_00` are executed and thus the behaviour is equivalent to the intended situation (i.e. the '`SECTION`' keyword after `X11_CONT_CHANGED` is not omitted). The real error occurs when the condition in `X01_00` is not true, and `PERFORM X12_ASP_CHANGED` is executed. After that statement terminates, the control-flow also reaches the statements in `X11_00` but this time these are executed unintentionally. Our findings were handed to the application manager of this system; he was particularly interested in the sections with a missing keyword, since he was surprised that such mistakes were not found earlier.

Although there were only four mines in this large system, the mines represent hazardous programming practices and errors that are a potential threat to the proper operation of the system. The low number of mines can be due to the low number of goto statements: only 12 in nearly 200,000 lines of code. Although mines are not always caused by goto statements, the figures on this system can be an indication that programming with less goto statements reduces the chances on mines significantly. In the next section, we show how the elimination of goto statements can help to fight minefields.

System 5 In System 5, we detected 413 mines in 61 programs. Most of the mines were fallthrough mines and these were all caused by unintended goto mines except for one; that fallthrough mine was caused by an omitted exit program statement, just like one of the mines in System 2. The number of goto Into and goto Out mines are not the same, because there were 5 jumps into the first section of the program, which was not performed itself, and there were 11 jumps from an inner range to a label in the outer range; hence, the jump originates in both the inner and outer range and its target is also located in the outer range. This is illustrated by the minefield in Figure 3.14. Such minefields cause portability problems, just like we observed in System 3: as soon as the jump is made out of the inner range, the continuation of the outer range is not active when certain compilers are used.

```
    PERFORM X01_COMMIT_CHECK
...
X01_COMMIT_CHECK SECTION.
X01_00.
    IF ASPS_REC = ZERO
       PERFORM X11_CONT_CHANGED
    ELSE
       PERFORM X12_ASP_CHANGED
    END-IF.
X01_99.
    EXIT.

X11_CONT_CHANGED.
X11_00.
    ...
X11_99.
    EXIT.

X12_ASP_CHANGED SECTION.
    ...
```

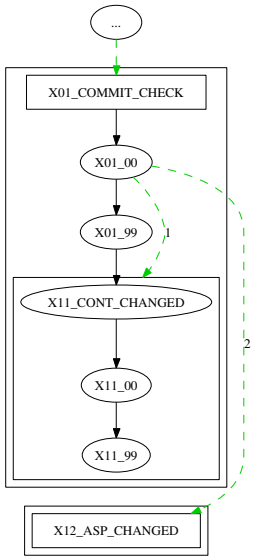

Figure 3.13: In System 4, an overlapping perform range was created unintentionally by an omitted a `SECTION` keyword after label `X11_CONT_CHANGED`.

```
     PERFORM RTC_AC.
...
RTC_AC SECTION.
10_01. ...
     IF PRS = 99
        ...
        GO TO 101_99.
10_10. ...
     IF PRS = 99
        ...
        GO TO 101_99.
     ...
     PERFORM 101_20 UNTIL PRS = 02.
     ...
     GO TO 101_99.
10_20. ...
     IF PRS = 99
        ...
        GO TO 101_99.
     ...
     IF PRS = 99
        ...
        GO TO 101_99.
     ...
10_99.
     EXIT.
```

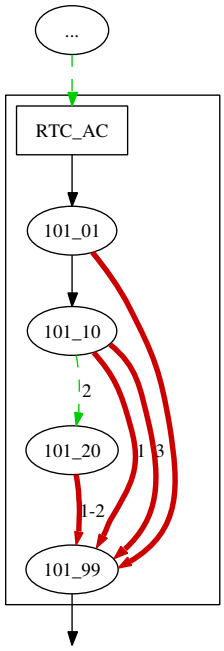

Figure 3.14: A Cobol minefield in System 5.

The rest of the goto mines were unintentional programming errors. The perform mines were explained as follows: 2 of them were caused by forgotten `SECTION` keywords (see the similar mines in System 4), and the other 151 were performed sections which performs some of its paragraphs as an internal subroutine. As we saw in Figure 3.14, one may be tempted to program a goto statement which jumps from the inner range to the outer range, which can cause problems.

Summary We applied our mine detector to five industrial Cobol systems and we were able to determine potential error-prone programming constructs: Cobol minefields. We observed that a mine hardly comes alone, since in almost all cases one mine introduced a number of other mines, creating a minefield. Hence, clearance of one mine will often result in clearance of other mines as well. Minefield clearance should always be done with care because a new error can be introduced. It is difficult to determine automatically whether a mine is programmed intentionally or unintentionally; therefore, one needs the help of a system expert. The majority of the mines was caused by unintentional programming errors, and can cause code to be executed in an unintended order. This leads to problems ranging from invalid output to a sudden breakdown of the system during production. In general, unintentional mines can be cleared but this should be done with care.

Still, a minority of the mines was programmed intentionally, and can hinder maintenance of the code. Although it can be difficult to deal with this type of mines, it is valuable for a system expert to map the minefields of a system and to be aware of their presence. We also observed that in a large system with only a few goto statements the number of mines can be significantly lower. Although we have little empirical data to establish a correlation, our findings can indicate a relation between the number of goto statements and the number of mines in a system. In the next section, we show how we can combat minefields by using goto-elimination and other restructuring techniques.

3.4 Demarcation of Cobol minefields

We detected and evaluated intentional and unintentional mines in five industrial systems. Due to problems that can arise from minefields, it can be desirable to clear them. However, mines can usually not be cleared that easily due to the imprecise semantics of the perform statement and the subjective meaning of the code. After all, it can be difficult to determine what the intention of the original programmer was, and the behaviour of a mine can be compiler-dependent. Nevertheless, we showed that a tool can detect and report minefields. In this section, we want to argue that the power of tools lies beyond the detection of minefields. We show how restructuring tools can help to fight minefields. A tool can simplify a complex minefield by demarcation of its perimeter, i.e. a tool can identify, mark, and separate areas that are considered safe. Then, a programmer is made aware of the code that should be approached with care.

Styles, standards and structures In our definitions of mines, mines involve the perform statement. Nevertheless, we believe that the primary sources of programming prob-

lems, such as mines, are the use of fallthrough logic and the employment of numerous goto statements to express programming logic. This may not be a novel thought, but we know from experience that there is a great deal of Cobol code in production that makes extensive use of this 'classic' style. We also believe that there are valid reasons for that, and we will explain the most common reasons here.

Many Cobol programs were initiated decades ago. At that time, Cobol lacked support for certain ways of programming, which are nowadays considered as structured. With the establishment of a new Cobol standard in 1985, new features for structured programming were added. Support for local loops was added by the in-line perform statement, replacing the need for a simulated loop by goto statements, and the explicit scoping of nested statements was another contribution that can improve the structure of programs. Furthermore, it was advised not to use certain language constructs, since these easily complicate the flow of control (e.g. self-modifying code using the alter statement). Then, as we discussed earlier, the 2002 standard added more features for structured control-flow, which are common in several newer programming languages. These included constructs for breaking out of local loops, and to prematurely end the execution of paragraphs and sections. The idea is that these features restrict the use of the goto statement to exceptional cases.

However, extension of the standard does not mean that the extensions are immediately incorporated in existing and new code, and that existing compilers provide proper support. Fortunately, many Cobol compilers have support for the 1985 standard by now, and a number of compiler vendors have started to implement the 2002 standard. Still, one can think of several reasons why a great deal of code is developed and maintained using a classic style with goto statements and fallthrough logic. One reason can be the common conception not to touch code that *works*. This is because a change means immediate costs and, more importantly, a change can introduce or reveal errors. So, unless it is imminent to apply a change (e.g. due to keep up with changing requirements or to correct an error), the modification of a program to keep up with the latest standard or to improve its internal structure has usually a low priority. Another reason can be that, when a programmer modifies code, it is common practice to adhere to the original coding style. Then, to avoid a mixture of styles, one may be forced to program using fallthrough logic and goto statements. In addition, experienced Cobol programmers may be more familiar, skilled, and productive when using the classic Cobol style; hence, they are not tempted to program with newer constructs. So, although there are ways in Cobol to reduce the need for fallthrough logic and goto statements, it is not trivial how to employ them effectively.

One approach to obtain structured programs in a consistent, reliable and cost-effective way is the use of automatic restructuring tools and to establish and enforce coding standards. Tools can detect violations of coding standards, and tools for restructuring Cobol code have existed for decades. These tools can automatically replace complex code by structured alternatives, resulting in a normalised control-flow without goto statements. However, there are several objections to the use of automatic tools. As a restructured program can be very different from the original program, it is likely that the main objections to use restructuring tools come from the original maintainers. Restructured code can become non-intuitive due to the normalisation of simulated constructs, the duplication of code, and the creation of meaningless flag variables and paragraph names [54]. In the

maintainers' view, their precious code is mutilated by some automatic process, operating beyond their powers. They are not eager to accept and maintain the result. Also, the argument not to change code that works can still be put forward. For example, a program with goto mines may behave differently after all goto statements have been removed. This is because the restructuring tool was tailored towards a specific semantics for the perform statement, which can be different from the implementation in the used compiler. Or a program has to be preprocessed to remove the mines before a tool can handle it; if this is done by hand it is prone to errors and can endanger the operation of the program. Nevertheless, for many programs, some form of restructuring is inevitable sooner or later. The key to successful restructuring is to know what, how, and when to restructure. We summarise some findings here:

- The need for restructuring can become evident when the code or its environment must change [171, 180]. For example, business requirements force the code to be changed, migrated, or reused. Then, restructuring is done to enable a system for change. Besides that, restructuring is then often accepted as part of the process since the code has to be changed anyway.
- The chances on acceptance of (automatically) changed code can be higher when the original maintainers are involved in the change effort, or when they are replaced [179] (e.g. retirement, outsourcing).

Although the internal quality of code is rarely a reason to modify code, we will illustrate how restructuring can aid to fight Cobol minefields; this adds another rationale for code restructuring. To do this, we present a restructuring approach in more detail. We do not state that this is the *best* way to deal with minefields; careful programming and other restructuring approaches can serve as well.

Restructuring of Cobol Sellink, Sneed and Verhoef [171] developed a method to restructure Cobol code which eliminates fallthrough logic by transformation of paragraphs into subroutines. The restructured program is divided into a main part and a subroutines part. The main part can contain goto statements and fallthrough logic, whereas the subroutines part contains only paragraphs that are free of fallthrough logic and goto statements. The primary goal is to extract as many paragraphs as possible and move them to the subroutines section, which means that a large part of the program is free of fallthrough logic and goto statements. Then, it is more obvious to add new code that does not interfere with the existing control-flow, which is in-line with common ideas on structured programming. In addition, it is the intention to alter the existing paragraphs as little as possible. Although the approach moves a great amount of code around, changes to the existing paragraphs are kept as small as possible. Since there are no flag variables or labels generated to normalise the control-flow, it is not always possible to remove all goto statements. The rationale is that some types of goto statements simulate constructs that were not available in Cobol at the time the code was written. The restructuring approach does not introduce features of the 2002 standard since these are not supported by all compilers; hence, some of the goto statements that remain after the restructuring can be replaced by variants of the extended exit statement. Sellink et al. presented an industrial Cobol program to illustrate the approach.

```
72-PT-MS SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC.
  MOVE 0 TO PT-M-TABLE-R.
7203.
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC < PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC = PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  SUBTRACT PERIOD-ACC FROM M-ACC.
  GO TO 7203.
7205.
  ADD PERIOD-ACC TO M-ACC.
  IF M-ACC > 72
    GO TO 7207.
  MOVE 1 TO M-TABLE(M-ACC).
  GO TO 7205.
7207.
  MOVE PT-M-TABLE-R TO PT-MS.
  MOVE 0 TO DEP-DATA IN TRA-NUM.
  IF P-TYPE
    GO TO 7299.
7209.
  MOVE 0 TO MIN-AMOUNT  IN TRA-NUM
            MAX-AMOUNT  IN TRA-NUM
            REMAIN-PERC IN TRA-NUM.
7299.
  EXIT.
```

```
72-PT-MS SECTION.
RESTRUCTURE-PAR.
  PERFORM 7201
  PERFORM 7203
  PERFORM 7205
  PERFORM 7207.
7299.
  EXIT.
  ... GOBACK.
72-PT-MS-SUBROUTINES SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC
  MOVE 0 TO PT-M-TABLE-R.
7203.
  PERFORM TEST BEFORE UNTIL ( M-ACC < 01 )
                OR ( M-ACC < PERIOD-ACC )
                OR ( M-ACC = PERIOD-ACC )
    SUBTRACT PERIOD-ACC FROM M-ACC
  END-PERFORM
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
  END-IF
  MOVE 1 TO M-TABLE(M-ACC).
7205.
  ADD PERIOD-ACC TO M-ACC
  PERFORM TEST BEFORE UNTIL ( M-ACC > 72 )
    MOVE 1 TO M-TABLE(M-ACC)
    ADD PERIOD-ACC TO M-ACC
  END-PERFORM.
7207.
  MOVE PT-M-TABLE-R TO PT-MS
  MOVE 0 TO DEP-DATA IN TRA-NUM
  IF NOT ( P-TYPE )
    PERFORM 7209
  END-IF.
7209.
  MOVE 0 TO MIN-AMOUNT  IN TRA-NUM
            MAX-AMOUNT  IN TRA-NUM
            REMAIN-PERC IN TRA-NUM.
```

Figure 3.15: Code before and after restructuring.

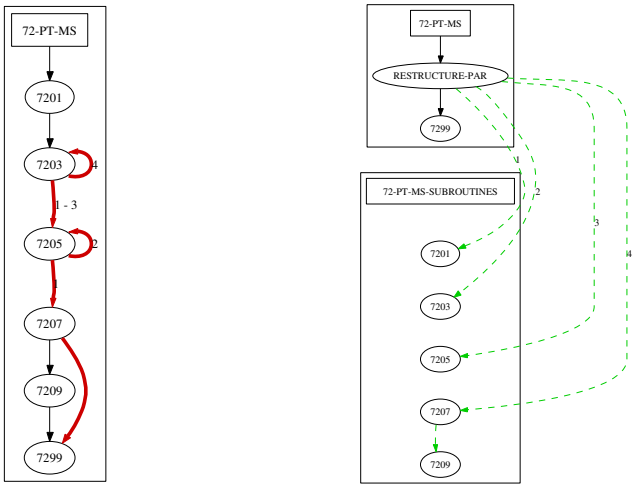

Figure 3.16: Control-flow graphs of the code from Figure 3.15.

```
     PERFORM C-DAT.
...
C-DAT SECTION.
...
C09. IF COUNT-BT > COUNT-1
        GO TO C99
     END-IF
     ...
     IF P-CODE = ZERO
        ...
        GO TO C09
     END-IF
     ...
C11. ...
     SEARCH TAB-LK
        AT END GO TO C99
        WHEN T-CODE = LK-1
             PERFORM C13
             PERFORM C15
     END-SEARCH.
C17. ...
     PERFORM C11
     ...
     GO TO C09.
...
C99.
    EXIT.
... GOBACK
C-DAT-SUBROUTINES SECTION.
C13. ...
     IF NOT ( AK-1 = ZERO )
        ...
     END-IF.
C15. ...
```

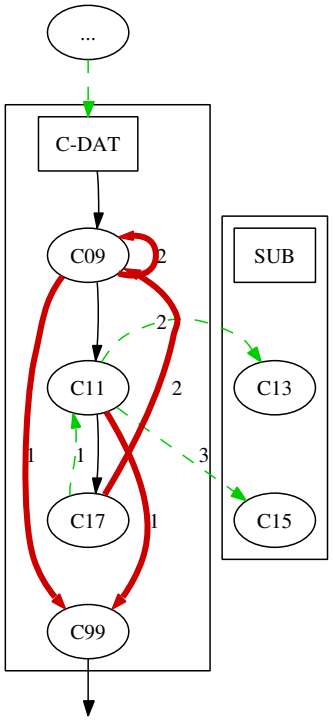

Figure 3.17: Demarcation of the minefield from Figure 3.12.

We continued their work to see whether the ideas are sufficient for restructuring large Cobol systems. In our case studies, which will be described in detail in Chapter 5, the restructuring approach was able to transform up to 80% of the code of a large system into subroutines (>2 million lines of code). This can be further increased if Cobol 2002 features are introduced. Hence, only 20% of the code remained in the main part, containing goto statements and fallthrough logic. To illustrate what this means, we discuss some of the production code that we restructured.

Figure 3.15 shows a small code snippet from restructuring effort, before and after transformation. The accompanying control-flow graphs are depicted in Figure 3.16. We briefly explain how the code has been restructured. First, a goback statement ensures that there is no fallthrough logic beyond paragraph `7299`. Then, the paragraphs are moved one by one. A moved paragraph is replaced by a perform statement that references the moved paragraph. The perform statement is added to the preceding paragraph. Existing perform

ranges are taken into account during restructuring to make sure that the behaviour of the program is not modified. This means that not every paragraph can be moved (e.g. due to the existence of mines or complex perform ranges). Each existing goto statement that references the moved paragraph is replaced by a perform statement and a goto statement. The perform statement references the moved paragraph and the goto statement jumps to the paragraph label after the performed paragraph. This way, the existing logic is preserved. Before a paragraph can be moved, it must be untangled from its surrounding paragraphs, which may require elimination of some goto statements. For example, when paragraph `7209` is moved to the subroutines section, `PERFORM 7209` is added to the end of paragraph `7207`. When the condition in the if statement in `7207` is complemented, the `PERFORM 7209` is placed in the if branch. Then, `GO TO 7299` can be removed. That way, `7207` has no external goto statements and can be extracted as well. This process continues until no more code can be moved or goto statements can be eliminated.

In our example, about 80% of the statements in the code is restructured into subroutines, which are performed from a main part that contains about 20% of the code. Paragraphs in the subroutines part have become loosely coupled blocks of code. One can rearrange, reuse or modify the separate blocks more easily since they are no longer tangled with each other. Moreover, one can safely add new paragraphs to the created subroutines part without interfering with the fallthrough logic between existing paragraphs.

Minefield demarcation We showed that a Cobol program can be divided into a main part and a subroutines part by restructuring. We now explain what such an approach can do to fight minefields.

With the described restructuring approach, in a restructured program, all remaining fallthrough logic and goto statements are located in the main part, and the subroutines part consists solely of performed paragraphs. Mines are not moved; they are recognised by the restructuring algorithm and left alone (we already explained that perform ranges are taken into account during the restructuring, in Chapter 5 this will be discussed in more detail). Hence, existing minefields remain in the main part of the program and the subroutines part is a safe area without mines. This means that minefields are automatically demarcated. Moreover, by moving the code surrounding a minefield to the safe area, we can shift its perimeter and reduce its size. We demonstrate this with one of the minefields we encountered in our case studies.

The minefield in System 3, which we presented in Figure 3.12, contains several mines. Although it is a complicated code fragment with perform, goto and fallthrough mines, we can transform some of its paragraphs into subroutines. This is shown in Figure 3.17. Two paragraphs are isolated from the minefield, and the perimeter of the minefield is shifted. This way, the size of the minefield is reduced and the size of the intricate code has been reduced to one paragraph. So, by restructuring a program, we can demarcate and simplify minefields.

To illustrate the demarcation of minefields and complex code in a large program, we present a case in which we restructured a large Cobol legacy application for an IT-service company.

Demarcation of a legacy application An IT-service company was requested by an insurance company to improve the maintainability of a large Cobol program, as part of a legacy portfolio modernisation project. It concerned a 15,000 lines mainframe program which was initiated more than 25 years ago. The company restructured the program by hand, reducing it to 10,000 lines. The reduction of 5,000 lines was mainly due to the removal of code that was considered to be no longer necessary. However, they wanted to restructure it further and realised that the use of automatic tools was inevitable for a fast and accurate result. They asked us to apply our restructuring tools to their program. The program we received contained only one section with 236 paragraphs, 4,690 statements, 501 goto statements, and 173 perform statements. It turned out that the program was initially developed with many goto statements, and later on people started adding perform statements. This resulted in about 30 paragraphs near the end of the program that were only reachable by perform statements, resembling a subroutines structure. Still, more than 200 paragraphs were tangled with intertwined logic, and we also detected two intentional goto mines. To get an impression of the complexity of the application, we generated a control-flow graph, which is depicted in Figure 3.18. With the electronic version of this thesis, it is possible to zoom in on the graph and read the paragraph labels. In Figure 3.19, we enlarge the top of the graph and we also show a snapshot of an application form for an insurance policy from the corporate website. The figure reflects the relation between the corporate website and the accompanying business logic in the Cobol program. The different types of insurances from the application form are represented by the names of Cobol paragraphs in the application.

At first sight, it may appear to be difficult to draw conclusions from the graph in Figure 3.18, but this is not true. We explain the graph here. There are 236 nodes, representing the paragraphs in the program. The location of a paragraph in the graph is influenced by its position in the source file and by the attached edges. This means that paragraphs at the top of the graph appear in the beginning of the file and that an edge between two nodes decreases the distance between them. Identical edges have been merged by our visualisation tool. From the top of the graph, about two-third of the graph is dominated by goto edges and some fallthrough edges. This red majority of the graph represents more than 200 paragraphs that are intertwined with unstructured goto and fallthrough logic. At the bottom of the graph, on the left-hand side, about one-third is covered with green perform edges, representing about 30 paragraphs with a subroutines structure. The conclusion that we can draw from this graph is that we are dealing with a complex and evolved program, containing a mixture of goto, fallthrough and perform logic.

We applied our restructuring tools to the program to increase the number of subroutine paragraphs and to demarcate a safe area from the intertwined goto logic. Some statistics of the original and restructured program are given in Table 3.4. We reduced the number of goto statements to 215 and we increased the number of perform statements to 454. The created main part consists of 43 paragraphs and the created subroutines part consists of 124 paragraphs. The total number of paragraphs was decreased because dead code was removed and some of the paragraphs were merged when the labels were no longer necessary. We generated a control-flow graph of the restructured program, which is shown in Figure 3.20.

Although it is not possible to distinguish the precise flow of the edges in the graph, the

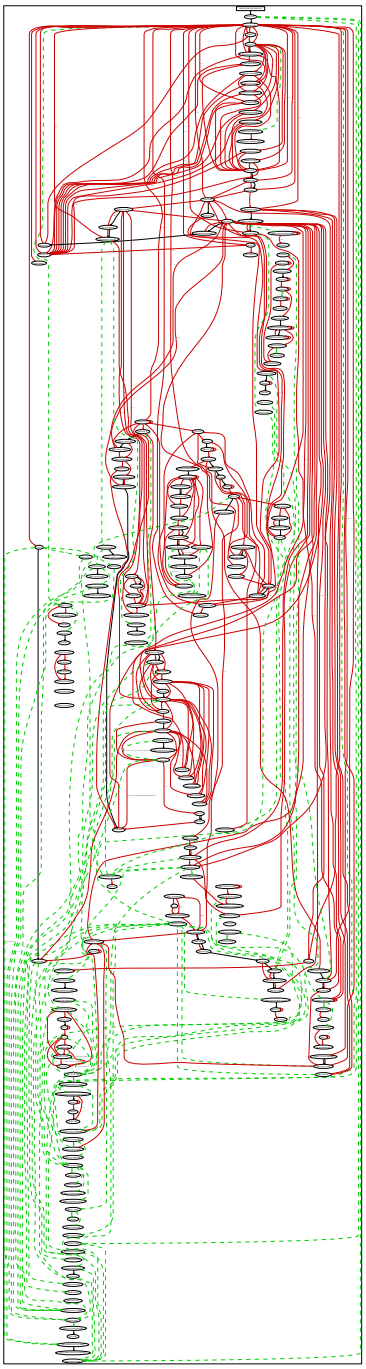

Figure 3.18: An evolved program

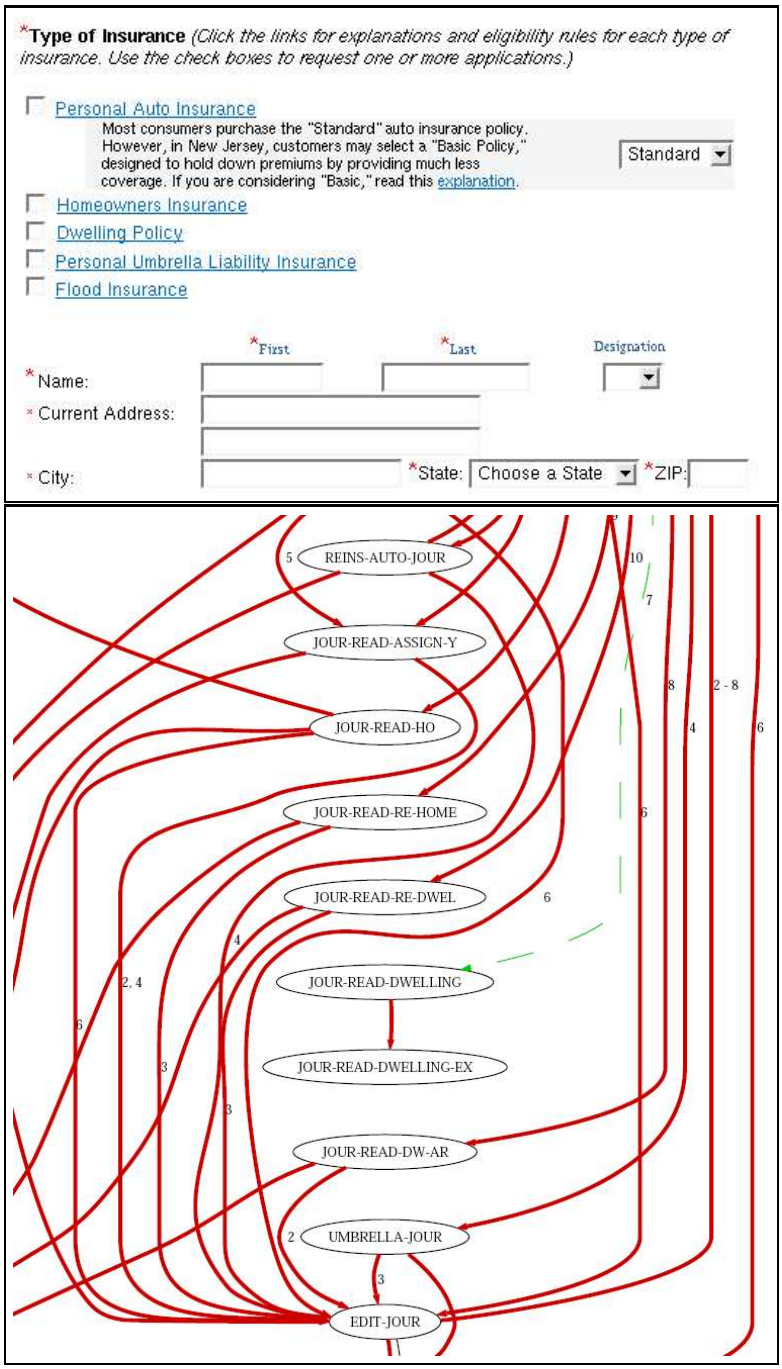

Figure 3.19: Part of the corporate website and a fragment of the control-flow graph of the accompanying business logic. The complete graph is depicted in Figure 3.18.

Table 3.4: Statistics of the restructuring effort.

	original	restructured	%
statements	4690	4119	- 12 %
goto statements	501	215	- 57 %
perform statements	173	454	+ 162 %
paragraphs	236	167	- 29 %
subroutine paragraphs	0	124	n/a
main paragraphs	236	43	- 82 %

figure clearly shows that the program has been restructured into a main part on the left-hand side and a subroutines part on the right-hand side. The remaining goto statements and the two mines are clustered together in the small main part, and the majority of the paragraphs can only be reached by means of perform statements in the subroutines part. The code in the subroutines part can still contain interrelated structures, but there are no goto statements and fallthrough logic among the paragraphs. Hence, there is a clear demarcation line between code containing the intertwined logic (fallthrough and goto statements) and code containing only (non-overlapping) perform statements.

The performance of the restructuring on this application can probably be improved by adjusting the tools. This means that we must add specific goto-elimination patterns that occur in the program. Our restructuring effort was a pro bono activity, so our time was limited and we did not adjust our tools to this particular application. Nevertheless, we were able to achieve a significant result and help the programmers with their modernisation project. After we restructured the program, we sent the code and the control-flow graphs to the company. They were pleased with the result and compiled and tested the program successfully.

Summary We argued that restructuring can help to fight minefields and other intricate code. We illustrated this with several examples, including an evolved legacy application. By using automatic restructuring tools, we divided a large program into two parts: a small main part that contains complex code, such as minefields, and a larger subroutines part that consists solely of loosely coupled blocks of code. The code in the main part should be approached with care, but the code in subroutines part can be moved, modified and reused more easily. This way, a clear demarcation line has been established between potentially error-prone code and code that is considered to be in a safe zone.

3.5 Conclusions

We investigated the behaviour of the Cobol perform statement in theory and practice, and we presented definitions for potential dangerous programming constructs in Cobol: mines. We analysed five industrial Cobol systems for minefields. We found that one mine often induces other mines, and that a mine can be programmed intentionally or unintentionally. Mines can cause sudden system failure, and can be hard to debug by hand. Fortunately, automatic tools can detect mines. Unintentional mines, representing

programming errors that may not yet have occurred, can be repaired by a system expert before they create havoc. The detected errors were caused by inconsistencies that had been introduced during maintenance. Intentional mines, representing intentional programming logic, are more difficult to deal with, as they often represent certain complex logic in a program. One can implement a tool for automatic removal of mines, but such a tool should be tailored towards a specific semantics. Instead, we argued that it is better to detect mines, as they can be an indication of intricate code that requires human inspection. Furthermore, we argued that code restructuring can aid to fight minefields, and we showed that an evolved Cobol program can be restructured into a relatively small isolated area with potentially error-prone code and a larger area containing only loosely coupled blocks of code. This way, a clear demarcation line is established between a safe zone and the dangerous minefields.

Acknowledgements We thank the anonymous reviewers of the journal "Software: Practice & Experience" for their valuable comments and suggestions for improvements. We are grateful to Steven Klusener and Chris Verhoef for their support and comments. Our thanks are also due to Peter Bol from Getronics PinkRoccade and Carl Iglesias and Grant LeMyre from Telecom Management Consulting Group for their cooperation. Gerbrand Stap supported us with the AS/400. We further thank others that provided us with industrial source code for the case study. We also acknowledge Wim Ebbinkhuijsen for his comments.

Road map We pursue the restructuring of Cobol programs in the Restructuring project in Chapter 5, where we develop and deploy automatic transformations to change-enable evolved legacy applications. This way, the modifiability of monolithic code is revitalised to allow and ease evolution. Before we address the code restructuring, which involve significant changes to a program, we present the mass maintenance process of a local change to thousands of programs in the next chapter. In the context of the Btrieve project, we present and deploy an approach to simultaneously carry out thousands of changes to thousands of programs in tens of intertwined systems.

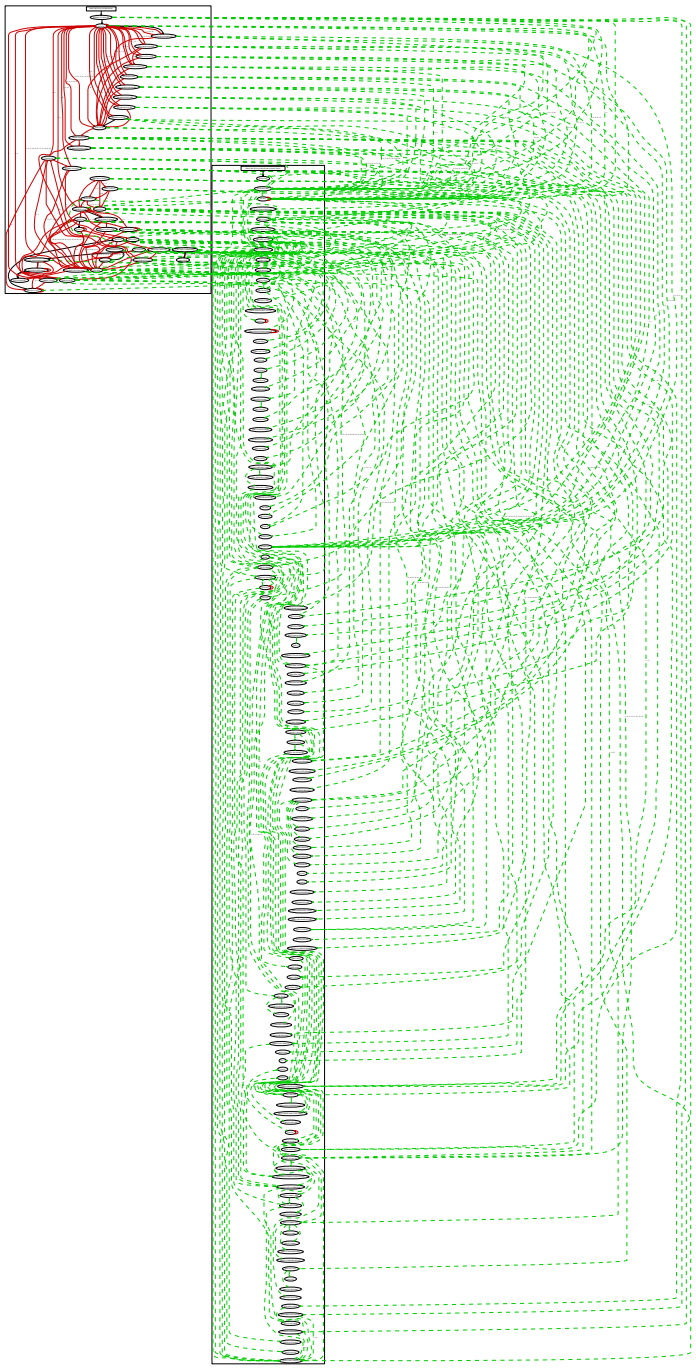

Figure 3.20: Automatic restructuring of the program from Figure 3.18.

Chapter 4

Automated mass maintenance of a software portfolio

A company in the financial services and insurance industry upgraded their database system to a new version. The database system was accessed by a portfolio of 45 interdependent systems, which consisted of nearly 3000 programs and covered more than 4 million lines of Cobol code. The upgrade affected almost all of the systems in the portfolio; hence, a simultaneous update of the involved programs was inevitable.

) In this chapter, we present an experience report on automated mass maintenance of a large Cobol software portfolio, involving simultaneous changes to thousands of programs. We discuss the advantages of automation and explain why a manual approach for such massive changes is infeasible. We upgraded the company's programs to the new database version using several automatic tools, and we performed an automated analysis supporting further manual modifications by the system experts. The automatic tools were built using a combination of lexical and syntactic technology, and they were deployed in a mass update factory to allow large-scale application to the software portfolio. The updated portfolio has been accepted and taken into production by the company, serving over 600 employees with the new database version. We will discuss the automated upgrade from problem statement to project costs. This chapter is based on: N. Veerman. Automated mass maintenance of a software portfolio. *Science of Computer Programming*, 62(3):287–317, 2006 [196].

4.1 Introduction

Software systems need to be updated from time to time. This can be regular maintenance, such as error corrections, as well as more structural modifications like conversions and migrations. The most well-known structural changes are Y2K and Euro, but there are many more of such *mass update projects* [108]. Hardware and software upgrades, expansion and conversion of data structures, platform and language migrations; these are all examples of changes which are sooner or later applied to any business-critical system.

These software projects share similar characteristics: the modifications must be made in a systematic way, i.e. in many places throughout a system, or even throughout an entire software portfolio. A problem with such massive changes is that they must be carried out all at once: they are interrelated and require a simultaneous update. Thousands of applications must be changed at approximately the same time. For example, the expansion of a data structure can affect a software system in numerous places. Before the updated system can be compiled and tested, all affected places must have been identified and properly modified.

Initially, when you look at the needed changes in many mass maintenance projects, they seem very regular. Often, one or two lines must be changed that can be found with a simple lexical tool like `grep`. So it is tempting to underestimate the harsh problems that lurk in the background, and many naive attempts to carry out such projects have failed. In fact, when the Y2K problem emerged, many thought exactly this: one can easily find the infected parts and change those. It turned out that such a simple approach is not sufficient to solve structural changes in large software systems. Therefore, the Gartner Group advises that any mass maintenance change for code volumes in excess of 2 million lines of code should be taken care of in an off-site software renovation factory [94, 109]. An example in [44, p248] illustrates that a seemingly simple modification — the removal of a single keyword — creates havoc when attempted in a naive way. In fact, as that paper shows clearly, even for a single keyword deletion it turns out to be harmful to use a simplistic approach, and properly controlled engineering methods are required for massive changes.

Nevertheless, popular belief is that minor changes by hand are not too complicated, but this is not true. One study found [86, p 333] that 55% of all one-line maintenance changes were erroneous on the first production run. Although this is an old reference, it stands until today. People are not good at consistently applying the same change many times, and many errors are introduced when changes are made manually.

But for the moment, let us assume that you would make massive changes by hand. Then consider the following reasons in favour of automation, taken from [117] and summarised:

- **Control** With an automatic approach, one knows exactly what needs to be changed, what is changed, how it is changed, in which order, and how to alter the changes themselves.
- **Consistency** People are not good at consistently applying rules by hand over and over again. If there are several variants of a modification, this is even more error-prone. An automatic tool applies changes consistently.
- **Completeness** With a manual approach, complications and variations of a problem statement are often overlooked in the initial phases, causing problems and delays later in a project, whereas an automated approach starts with properly defining the problems before making changes, and can be adjusted more easily.
- **Repeatable** A mass modification often has to be applied several times to different versions or parts of a system or portfolio. For instance, one time because some of the source code is missing, one time for testing, one time because the requirements

changed, one time to a small portion of the system, one time to the entire system or portfolio, and so on. Once an automatic tool has been developed, this can be done at low costs. With a manual approach, on the other hand, this is infeasible. So what appears to be a simple change that can be done at one time by hand turns into a complicated problem when approached in a naive way.

- **Execution time** Once automatic tools have been developed, the actual application time of the tool is relatively short compared to a manual approach. Especially when a mass modification is carried out during regular maintenance, the application time can be limited to one night or one weekend.

- **Reuse & customisability** If there are different versions of a system in production, different versions of an automatic tool may be needed. These tools often provide a high degree of reuse and flexibility in such situations. If during a project the requirements change, automatic tools can often be customised quickly. With a manual approach, it is error-prone to undo part of a modification.

As also noted in [117], a fully automatic solution is not always feasible and cost-effective. The amount of automation depends on several things; for instance, how often should a change be made, are interactive steps required to carry out the task, can a change be captured in a feasible number of code patterns, or can an automatic analysis support a manual modification by system experts.

In this chapter, we report on an automated mass maintenance project that was carried out by our team. We analysed and updated an entire software portfolio using automatic tools that were tailored to the problem. The updated portfolio was compiled, tested, accepted and taken into production by the company. We describe the entire process from problem specification to implementation and testing. We elaborate on the technological and economical aspects, and we illustrate the value of an automated approach in this real-life case.

The Btrieve project: massive maintenance of a software portfolio A company in the financial services and insurance industry upgraded their database management system to a new version. The Btrieve database management system from Pervasive Software [162] was used for keeping the customer records, as well as for the accounting. Over 600 employees within in the company used the system on a daily basis. Access to the database was provided through Micro Focus Cobol [150] systems on a Windows platform, which were initiated in the 1980s and continually modified and enhanced.

Due to the growth of the number of customers as well as other factors the size of the database has increased a great deal since it was originally initiated. This growth was reflected in the performance of the applications and the company decided to upgrade their database management system to a new and faster version of Btrieve, Pervasive.SQL [162]. This new version was not entirely backward compatible to the old version due to some changes in the language interfaces. In the software portfolio, several of the existing calls to the new database version were unsuccessful and had to be altered. The changes to the language interfaces were documented by the vendor of the database management system. Hence, the system experts of the Cobol applications knew, in theory, how to update the

calls to the database. However, there were nearly 50 thousand calls to the database spread over the software portfolio of 45 interrelated systems, covering more than 4 million lines of code (MLOC). For the company it would be risky to carry out such a drastically scattered portfolio-wide change in the usual way, because it is far from the day-to-day routine of normal maintenance. In fact, this type of systematic change is subject to active industrial research in projects [13, 53, 75, 77] and conferences [20, 143, 187, 205]. The company also realised this, and sought our assistance in solving this problem.

We were asked to carry out massive modifications to their entire software portfolio. Several small changes had to be made in many places to a large amount of source code. In addition, the company requested a portfolio-wide analysis to detect possible status code errors, such that the system experts could resolve the errors by hand. The problem statement for the Btrieve project was clear at the start: three modifications concerning five different database operations had to be inspected and possibly altered. In three of the database operations, a variable had to be substituted. If the new variable was not yet declared, its declaration should be added to the program. In the other two operations, one of the arguments of the database call had to have a minimum size which should be calculated from one of the other arguments. To illustrate the case, here is an example of a Btrieve database call in Cobol:

```
call BT "__BTRV" using b-cre, b-status, pos-block,
                       data-buf, data-bufl, key-buf, key-0.
```

The database call has seven arguments: an operation code, a status code, a position block, a data buffer, a data buffer length, a key buffer, and a key number. The operation code determines what action must be performed by the database (`b-cre` means create). The status code indicates whether any errors occurred during the operation. Position block is used to store the file structure and positioning information associated with certain operations. The data buffer is used to transfer data to and from a file, the data buffer length indicates the size of the data buffer. The key buffer and key number are used for searching in the database, as well as setting the file mode. So, such calls were scattered through the entire software portfolio.

The size of the portfolio and the type of the modifications are typical for a mass modification project: many small modifications are spread over a large amount of interdependent code. At this point, some people may think of such problems that you should assign 4 programmers to it and let them do the job in a week. Again, it is very tempting to think in this way, but it is bound to fail as some practitioners have found out the hard way. As it turned out in the Btrieve project, the company had tried to do some of the changes earlier by hand, which failed and complicated future changes even more. Eventually, this earlier attempt led to the cancellation of one of the changes near the end of our project. We were able to undo this modification without any problem, but that is not so easy to do by hand. Although scrupulous version management can roll back to the earlier situation, there is no guarantee that other useful changes are still in place. This illustrates that mass maintenance projects must be approached with care.

Related work The Btrieve project follows naturally from a line of research and practice done in the area of automated software analysis and modification. We will explain that here, starting with a review of some earlier work.

Large software portfolios and problems of massive software modifications, such as Y2K and Euro, are discussed in [198]. The proposed solution to get a large software portfolio under control is to use a software renovation factory equipped with automatic tools. Fundamentals for software renovation factories and system renovation are laid out in [33, 173, 199], and prerequisites for analysis and conversion tools for large software systems are discussed in [34]. A quick introduction into software renovation is given in [70], definitions of a software renovation factory are given in [94, 109], as well as in [83]: a software renovation factory is a set of software renovation assembly lines, a software renovation assembly line is an ordered set of (renovation) automated functions.

The purpose of a software renovation factory is to handle the transformation of massive amounts of code, using (renovation) tools such as parsers, analysers, transformations and unparsers [44]. Also in [44], an approach is presented for the generation of tools for software renovation factories: the ASF+SDF Meta-Environment [45, 48, 112]. The ASF+SDF Meta-Environment is a development environment for the automatic generation of interactive systems for manipulating text written in a formal language, thereby supporting the formalism ASF+SDF to specify grammars and transformations. In the Btrieve project, we used this environment to generate tools for updating the software portfolio, and we discussed this technology in Chapter 2. In Section 4.3, we briefly explain the details of the technology which are necessary to understand the examples in this chapter.

The development of such meta technology is motivated by a number of industrial projects and problems from the past few years. In [31], early industrial applications are presented. The technology was applied to prototype a domain-specific language, and the renovation of Cobol is discussed. Later on, in [171, 194], renovation projects of industrial legacy Cobol are presented with a software renovation factory approach. Several Cobol systems were restructured using automatic transformations. One of these projects is discussed in Chapter 5. In [52], a 110 KLOC Cobol system was migrated from Cobol85 dialect to Cobol74 by replacing a number of language constructs using a software renovation factory. In [42], a software renovation factory was implemented for control-flow normalisation for Cobol.

In a recent article [117], architectural modifications to deployed software systems are discussed. An architectural modification project is described where data items for product codes are expanded by one digit in a Cobol system of about 90 KLOC. At first sight, such a project may not look like an architectural modification. However, the product codes were scattered throughout the system's structure, and the seemingly simple change of one digit required a pervasive change to the system. A definition for software architecture of deployed software is given [117, p166]:

> *The software architecture of deployed software is determined by those aspects that are the hardest to change.*

The authors conclude that software architecture is about the immutable aspects of a software system. Possible candidates for such aspects include implementation languages, platforms, development environments, APIs and database schemas. Whether a particular aspect in a given system is part of the system's architecture depends on how difficult it is to change the particular implementation of the aspect. However, to determine how difficult it is to change something is subjective. One can think in terms of the involved risks, the dependencies of the software components to be modified, the impact on the ongoing operation and maintenance of the software, or even the impact of changes on existing business processes. Hence, architectural modifications can be hazardous and expensive to carry out, but automated support can significantly reduce the required effort and risks for major operations. According to the above description, the Btrieve project would qualify as an architectural modification. We had to modify the locations that interfaced with the database system. There were nearly 50 thousand places where the database system was accessed scattered throughout 45 intertwined systems; hence, a structural modification to the interface locations can be considered to be a major operation.

An example of a portfolio-wide analysis effort is an analysis to automatically count function points from the source code of an entire IT-portfolio [115]. This is a prerequisite to calculated decision-making regarding proposals for modifications that impact an entire IT-portfolio, as in our case. In [200, 201] a quantitative approach is described to provide insights into the costs, durations, risks, returns, and financing issues for such changes. In [118], an analysis was performed to estimate the impact of the expansion of bank account numbers. It was observed that the use of automatic detection and modification tools reduces the costs of such an expansion significantly.

In [179], risks involved in reengineering projects are discussed. Based on the experience of 13 projects, five reengineering risks were identified. According to the article, the risk that is most difficult to deal with is the rejection of the results by the programmers who maintain the original system. It is mentioned that programmers tend to reject results with which they have not been involved. In the Btrieve project, to avoid rejection, the programmers (or system experts) were involved at the start of the project, and they were consulted several times during the project. The final results were delivered in agreement with the programmers; hence, the results were not rejected.

There is a great deal of related work in the field of automated software transformations and maintenance. Recently, a special issue of *Science of Computer Programming* on program transformation [127] was published containing a wide range of papers on program transformation. Several papers from the program transformation community appeared. A paper related to the Btrieve project is the paper by Ward [211], which describes an automated migration of 544 KLOC assembler code with the FermaT system. Advantages of automated reengineering are discussed: scalability, customisability, low impact on ongoing development, low resource requirement, and other advantages. Although the paper deals with an automated transformation project, the focus is on the FermaT technology and the project is not described as extensive as the Btrieve project, nor it is of the same size.

Another technology for automated maintenance purposes can be found in Baxter's work [15, 16]. The described technology, DMS, was applied in several large-scale maintenance projects. Cordy's work on TXL [62, 63] was deployed for Y2K and other prob-

lems. Sneed has a workbench [178] which was applied in large-scale renovation projects. The TAMPR transformation system was also used for automated maintenance tasks, such as Y2K analysis and conversion [28, 29]. AnnoDomini [76] is a commercial Y2K conversion tool for Cobol programs. In addition, a number of companies deploy automated maintenance tools [55, 164, 175, 176, 184]. In the Btrieve project, we used the ASF+SDF Meta-Environment [45, 112], which has also been used in several other projects (e.g. [31, 42, 41, 52, 171, 194]).

Contributions In related work, significant work has been done in the area of automatic analysis and manipulation of industrial software. The Btrieve project synthesizes the effort and experiences of previous projects to carry out a real-life large-scale maintenance case: the automated mass maintenance of an entire Cobol portfolio.

4.2 Problem statement of the Btrieve project

The requested modifications and analysis were driven by several constraints that the new version of the database management system imposed on the Cobol applications. The new version was not entirely backward compatible, and when the system experts tested the applications using the new version they encountered various unexpected problems. For example, in the new version it was obligatory for some database operations to calculate the length of one of the call arguments and then supply it to the database call. In the old version, this was not required. Also, the file cursors were handled differently in the old version. Using the new version, applications caused status codes which indicated an unsuccessful database operation. Therefore, each access to the database that was affected by the upgrade had to be extended to handle the erroneous status code.

4.2.1 The three requested modifications and the analysis

The problem statement was formulated in consultation with the system experts. We were commissioned to do three modifications and an analysis, which was used to aid the system experts in making manual changes to the portfolio. We briefly describe the problem statement, and then we elaborate on the problems and requested solutions, giving several examples.

- **Key-0** modification: For all reset (`b-res`) and unlock (`b-unl`) database operations, the key number argument should be (changed to) variable `key-0` with value 0. If this constant is not present in either the working storage section of the program or in one of the included copybooks, the following declaration should be added to the working storage section of the program:

  ```
  77 key-0 pic 99 comp-0 value 0.
  ```

- **Dbl-4** modification: For all get position (`b-gpo`) database operations, the data buffer length argument should be (changed to) variable `dbl-4` with value 4. If this constant is not present in either the working storage section of the program or in

one of the included copybooks, the following declaration should be added to the working storage section of the program:

```
77 dbl-4 pic 99 comp-0 value 4.
```

- **Data-length** modification: For all status (`b-sta`) and create (`b-cre`) database operations, the value of the data buffer length argument should be at least the length of the data buffer. If this value is less than the actual length when the call to the database system is made, the following statement should precede the call:

  `move` <length of data buffer> `to` <data buffer length variable>.

- **Cursor** analysis: A complex analysis had to be carried out to detect conflicting file cursors. A program that reads from a data file alters the cursor position in that file, and when another program reads the same file the cursor is also altered. This can lead to conflicts when the first program tries to read the file again. The error occurs only with the new database version because the file positions are stored differently from the old version. To detect potential problems, the analysis had to detect loops in which certain database operations are performed on a file and also another program is executed which operates on the same file. The results of this analysis are used to aid a manual change by the system experts to avoid the conflicting cursor positions.

A first impression of the Key-0 and Dbl-4 modification is that they involve simply replacing a variable in one line of code and perhaps declaring a new variable. A naive approach to solve part of these problems would be to start with adding the variables to every program, and let a compiler diagnose the unused and redundant variables. This would have to be done after all database calls have been updated adequately in some way, and would require much effort since all reported variables must be removed. The most prominent objections to such an approach are the feasibility and costs. Since, according to Gartner [94, 109], mass maintenance for code volumes in excess of 2 MLOC should be taken care of in an off-site renovation factory, a suitable compiler is often not available. Although the owner of the system has a compiler and test environment, it is expensive to install the same environment at the renovation site. We refer to [78] for figures on costs for compiling and testing large systems.

So, in the Btrieve project, three modifications had to be carried out, concerning five different database operations that had to be examined and possibly altered. In addition, a complex analysis had to be done. Next we describe each modification and the analysis in more detail, with some code samples from the portfolio before and after modification. In the code, each modification is annotated by a comment line.

Key-0 Figure 4.1 shows some code before and after the Key-0 modification. There is a call to the database system with a reset operation and the current key number argument is `acc-1`. This should be changed to `key-0`, and the `key-0` declaration should be added to the data declarations if it is not present in the working storage section or in one of the copybooks.

```
       ....
       copy h:\rlo\btrv.cpy.
       copy h:\euro\rlo\eff001.rlo.
       copy h:\rlo\edb001.rlo.
       ....

       no99.
           call BT "__BTRV" using b-res, b-stat, ec-edb001,
                edb001-record, dbl-edb001, edb001-pos, acc-1.
```

```
       ....
       copy h:\rlo\btrv.cpy.
       copy h:\euro\rlo\eff001.rlo.
       copy h:\rlo\edb001.rlo.

* Btrieve 7.9 modification: add declaration
       77 key-0 pic 99 comp-0 value 0.
       ....

       no99.

* Btrieve 7.9 modification: KEY-0
           call BT "__BTRV" using b-res, b-stat, ec-edb001,
                edb001-record, dbl-edb001, edb001-pos, key-0.
```

Figure 4.1: Key-0 modification.

```
       ....
       77 question1 pic x(45) value "Change (Y/N) ?".
       ....

       GPO-TET204 SECTION.
       gpo01.
            call BT "__BTRV" using b-gpo, b-stat, ec-tet204,
                                   adr-3, dbl-3, buf-tet204, key-0.
```

```
       ....
       77 question1 pic X(45) value "Change (Y/N) ?".

* Btrieve 7.9 modification: add declaration
       77 dbl-4 pic 99 comp-0 value 4.
       ....

       GPO-TET204 SECTION.
       gpo01.

* Btrieve 7.9 modification: DBL-4
            call BT "__BTRV" using b-gpo, b-stat, ec-tet204,
                                   adr-3, dbl-4, buf-tet204, key-0.
```

Figure 4.2: Dbl-4 modification.

```
       ....
       77 data-bufl          pic 9(05) comp-0 value 103.
       01 status-block.
          03 etq-rl           pic 9(2) comp-0.
          03 etq-ps           pic 9(2) comp-0.
          03 etq-rsv          pic x(4).
          03 key-specs occurs 2 times.
             05 key-position  pic 9(2) comp-0.
             05 key-length    pic 9(2) comp-0.
             05 key-flag      pic 9(2) comp-0.
             05 filler        pic x(10).
          03 filler           pic x(75).
        ....

       0191-00.
          call BT "__BTRV" using b-sta, b-stat, ec-tet248,
                   status-block, data-bufl, buf-tet248, key-0.
```

```
       ....
       77 data-bufl           pic 9(05) comp-0 value 103.
       01 status-block.
          03 etq-rl           pic 9(2) comp-0.
          03 etq-ps           pic 9(2) comp-0.
          03 etq-rsv          pic x(4).
          03 key-specs occurs 2 times.
             05 key-position  pic 9(2) comp-0.
             05 key-length    pic 9(2) comp-0.
             05 key-flag      pic 9(2) comp-0.
             05 filler        pic x(10).
          03 filler           pic x(75).
        ....

       0191-00.

* Btrieve 7.9 modification: data-length assignment.
          move 115 to data-bufl
          call BT "__BTRV" using b-sta, b-stat, ec-tet248,
                   status-block, data-bufl, buf-tet248, key-0.
```

Figure 4.3: Data-length modification.

Dbl-4 The code snippet in Figure 4.2 shows some code before and after the Dbl-4 modification. There is a call to the database system with a get position operation. The data buffer length should be replaced by `dbl-4` and its declaration should be added to the working storage section or it should already be declared in one of the copybooks. In this case, the declaration is added to the program.

Data length In the code fragment in Figure 4.3, an example of the Data-length modification is shown. Each `pic 9(2) comp-0` declaration takes two bytes, `pic x(4)` takes 4 bytes, `pic x(10)` takes 10 bytes, `pic x(75)` takes 75 bytes, and the `05` level records occur twice so the total size of data buffer `status-block` is 115 bytes. The initial value of data buffer length variable `data-bufl` is 103 and thus too small. A `move` statement with the correct value is added.

Cursor analysis Some of the programs showed status code errors while using the new database version. This was caused by different programs that modified the cursor position in the same file. In the old version, this error did not occur since the file positions were stored in a different way. Therefore, we were asked to do an analysis that identified these programs and the code that caused the errors. The system experts then manually changed the code in the reported programs to store and retrieve the file positions properly.

We did not deploy an automatic modification tool for this problem because the system experts expected that a relatively low number of programs would need modification. That would not be cost-effective to automate. They estimated this by looking at the number of programs that showed the status code errors. The actual number of programs that need modification can deviate from this estimate, since several errors can be repaired by patching a single piece of code, and there could be programs that had not yet shown this error at that time. Despite the advantages of an automated approach, it was considered not be cost-effective to automate the cursor modification. The system experts decided to carry out the modification by hand using our analysis results. However, we want to mention that a manual approach for this modification can introduce inconsistencies that may cost more in the long run. Even if only 10 programs need to be modified, the code for storing and retrieving the cursor positions will be copied, pasted and slightly modified by hand. Then, without thorough testing, inconsistencies can be introduced. This is similar to the minefields we discussed in Chapter 3, which will cause problems sooner or later.

To carry out the cursor analysis, we had to identify the programs containing a loop in which both a cursor position of a file was changed and a different program was called that also changed the cursor position in the same file. The error occurred when the first program tried to access the file with the modified cursor position again. We were asked to search for such logic in loops, since the error occurred when the first access operation was repeated, and that operation expected that the file cursor was unchanged. Using our results, the system experts modified the programs by retrieving the cursor position before calling the second program, and restoring it after the second program terminated.

In the code snippet in Figure 4.4, an example of such a loop is shown. `PROGRAM1` contains a loop in `SEARCH-POLICYHOLDER` retrieving a record from `file001`, and subsequently `PROGRAM2` is called via two performed sections. `PROGRAM2` also changes

```
********** PROGRAM1 **********

        SEARCH-POLICYHOLDER SECTION.
           ...
        sph10.
           if tabcounter = 3
                   go to sph99.
           move file001-regnr to kw2-number.
           if not kw2-number1 = kw-number1
                   go to sph99.
           perform fill-table.
        sph20.
           call BT "__BTRV" using b-gne, b-status, fb-file001,
               file001-record, dbl-file001, buf-file001, key-0.
           if b-status = 9
                   go to sph99.
           if not b-status = 0
                   move "FILE001" to b-code
                   perform error-handling.
           go to sph10.
        sph99.
           exit.

        FILL-TABLE SECTION.
           ...
           perform get-part-av.
           ...

        GET-PART-AV SECTION.
           ...
           call "PROGRAM2" using ui-number, start-date,
                start-date, code-kw, code-av, code-kl, coll,
                dat-wz-kw, dat-wz-av, dat-wz-kl, dat-st-wt.
           ...

********** PROGRAM2 **********

        DETERMINE-POLICY-RIGHT SECTION.
           ...
           call BT "__BTRV" using b-gle, b-status, fb-file001,
               file001-record, dbl-file001, buf-file001, key-1.
           ...
```

Figure 4.4: Two programs altering the same file cursor.

```
*********** PROGRAM1 ***********

       SEARCH-POLICYHOLDER SECTION.
          ...
       sph10.
          if tabcounter = 3
                  go to sph99.

* Get position of file001 and store in adr-1 for status problem
          perform gpo-file001.

          move file001-regnr to kw2-number.
          if not kw2-number1 = kw-number1
                  go to sph99.
          perform fill-table.

* Restore position in file001
          perform gdi-file001.

        sph20.
          call BT "__BTRV" using b-gne, b-status, fb-file001,
              file001-record, dbl-file001, buf-file001, key-0.
          if b-status = 9
                  go to sph99.
          if not b-status = 0
                  move "FILE001" to b-code
                  perform fout-afhandeling.
          go to sph10.
        sph99.
          exit.

* Subroutine for retrieving the position
        GPO-FILE001 SECTION.
        gpo01.
          call BT "__BTRV" using b-gpo, b-status, fb-file001,
               adr-1, dbl-4, buf-file001, key-0.
          ...

* Subroutine for restoring the position
        GDI-FILE001 SECTION.
        gdi01.
          move adr-1 to file001-record.
          call BT "__BTRV" using b-gdi, b-status, fb-file001,
              file001-record, dbl-file001, buf-file001, key-0.
          ...
```

Figure 4.5: Program 1 from Figure 4.4 after cursor retrieval (Program 2 and two sections from Program 1 are not shown).

the position of `file001` (the same position block `fb-file001` is used). When `PROGRAM2` terminates, control-flow is passed back to `PROGRAM1` which can then access `file001` for the second time but with a modified position block. In the new database version this causes an error since position blocks are stored differently. So, such loops must be detected and the position block preserved and restored when the call to the second program returns. This is shown in the code snippet in Figure 4.5: two subroutines are added, one to save the position cursor in `adr-1` by a get-position operation (`b-gpo`) and one to restore the position cursor after the call with a get direct operation (`b-gdi`). Note that `PROGRAM2`, `FILL-TABLE SECTION` and `GET-PART-AV SECTION` are not shown in this figure.

4.2.2 What about semantics and correctness?

One of our aims is to perform any requested change quickly and accurately. Therefore, this paper is not about how to make the *best* change but how to make the *requested* change. In a mass maintenance project, requested modifications may not always appear to be sensible and sound at first sight. For instance, the Cobol85 to Cobol74 project [52] was about changing a system to an *older* Cobol dialect. Such a request may seem unusual if you are not working in industry, but real-life software systems require non-trivial changes, which can be neither correctness nor semantics preserving. Of course, when you make massive changes, you have to be aware of the intricacies that can appear in a program or portfolio. For example, if you do dataflow analysis for a Y2K change, you must know which statements affect the dataflow and in what way. On the other hand, for some problems it is easier to check them beforehand and, if they appear in the code, alter the code in advance instead of overloading your automatic tool to deal with complicated exceptions. We illustrate this with the modifications that were requested in the Btrieve project.

Some of the requested modifications required adding a variable to a program. So, if a variable of that name already exists but is declared or used differently, it could cause errors. In our case, the company had coding standards for these variables that restricted their use. With a simple lexical tool we checked in advance that the variables were not used outside the scope of these standards. This way, we did not have to take such exceptions into account when we developed the automatic tools. A similar situation can occur with the Data-length modification, which is also not correctness preserving in all cases. We had to calculate the length of the data buffer and then store the result in the data buffer length variable. Such a modification can cause errors elsewhere in the program. If the variable is used after the database call and a specific value is expected, the behaviour of the program can be different and errors may occur. Again, the presence of such constructs was quickly checked with the use of simple lexical tools. If one of these issues had appeared in the programs, depending on the issue, we would have preprocessed the code or implemented it in the automatic modification tools.

Assume that we are somehow able to develop an approach that can deal with all possible exceptions that can occur. In order to prove semantic and correctness properties of the changes, we need a formal semantics of the involved technologies. For a language like Cobol there is no single semantics because of the diverse variants, which is illustrated in [130, p83]. In the Minefield project in Chapter 3, we found that a basic language con-

Table 4.1: Statistics from the software portfolio

	amount	lines of code
Systems (programs+copybooks)	45	4,470,167
Programs	2,954	2,750,122
Total copybooks	19,444	1,720,045
Copybooks with unique names	7,195	828,248
Copybooks with unique content	9,011	1,022,139
Total database calls[a]	48,614	97,228
Relevant database calls	3,478	6,956
Systems with relevant database calls	44	4,466,085
Programs with relevant database calls	1,930	1,023,978

[a]Lines of code for the calls is based on two lines of code for each call

struct like the procedure call has different implementations in Cobol, and that there exist business-critical systems whose operation relies on the behaviour of a particular implementation. In addition, there is no formal semantics available for the Btrieve database technology. Hence, for a formal approach we should somehow obtain the specific semantics involved in the Btrieve project. We consider this to be infeasible because it is too time consuming for an industrial project.

We argued that semantics and correctness do not play a prominent role in mass change efforts, that requested changes from industry can be non-intuitive, and that exceptions to changes should be checked beforehand and removed instead of dealing with them in the automatic change tools. These are some of our findings with mass modification efforts.

4.2.3 Code exploration

Our next step was to identify the occurrences of the relevant database operations through the entire portfolio, to determine our approach, and to estimate the effort for the update. To do this, we did simple code explorations. We measured the size of the portfolio and estimated the number of database calls that had to be examined and possibly modified. This was done with basic UNIX tools like `grep`.

Table 4.1 shows some statistics of the portfolio. The size of the portfolio was about 4.5 million lines of Cobol code in 45 systems. There were 2,954 programs accounting for 2.8 million lines of code. On top of this there were 19,444 copybooks (Cobol include files) with another 1.7 million lines of code. As it turned out later, many of the copybooks were actually a clone of the same copybook; there were 7,195 unique copybook names, and 9,011 with a unique content taking up 1 million lines of code. We will give more detailed statistics in Section 4.4 (e.g. number of programs and lines of code in each system). In the entire portfolio, there were 48,614 calls to the database system. The database calls that had to be examined and possibly corrected appeared 3,478 times: in 44 of the 45 systems and in 1,930 of the 2,954 programs. Below we show how we did a quick analysis to estimate the impact of the change. We use `grep` to retrieve most of the relevant calls to the database, using five simple search patterns with the database operations. The `-l` option of `grep` reports all files that match. Some exceptions to these patterns may be

missed with this simple query, but this approach is powerful enough to make an estimate.

```
Number of relevant calls to the database:

$ grep -c -i -e 'call BT "__BTRV" using b-res,'
             -e 'call BT "__BTRV" using b-unl,'
             -e 'call BT "__BTRV" using b-sta,'
             -e 'call BT "__BTRV" using b-cre,'
             -e 'call BT "__BTRV" using b-gpo,' *.CBL
    3478

Number of relevant files:

$ grep -l -c -i -e 'call BT "__BTRV" using b-res,'
                -e 'call BT "__BTRV" using b-unl,'
                -e 'call BT "__BTRV" using b-sta,'
                -e 'call BT "__BTRV" using b-cre,'
                -e 'call BT "__BTRV" using b-gpo,' *.CBL
    1930
```

These numbers indicated that 3,478 calls in 1,930 of the programs needed to be examined and possibly updated. Furthermore, the 19,444 included copybooks had to be taken into account since they can contain declarations of variables and constants needed for the update of the database calls. So, at most less than 1% of the code was impacted (3,478 calls $*$ 2 LOC per call / 4.5 MLOC) but it was spread out over 98% of the systems and 65% of the programs. This characteristic was also found in a large-scale maintenance project where bank account numbers needed to be converted from 9 to 10 digits [118]. In that application, it was found that the impact of the change would affect 75-100% of the IT-systems, while only 2-8% of the source files even contained 9-digit numbers.

We performed some more queries to get an impression of the interdependencies among the systems. Using `grep`, we retrieved all call statements in each system. This resulted in about 110 thousand call statements, including the calls to the database system. Some of the calls were dynamic, which means that the called program can be determined at run-time; we ignored these calls. Furthermore, there were calls to general-purpose utilities, which we also filtered. We identified to which system a called program belonged by searching each system for a filename that matched the called string. If no file was found, the call was ignored. In total, we found about 23 thousand calls between the portfolio's systems, and about 5 thousand internal calls (i.e. calls within a system). We omitted the internal calls. Using the retrieved data, we established a call relation: a system has an outgoing call if there is one or more call statements that reference a program in another system. Note that this relation does not indicate how many individual call statements relate two systems. We visualised the call relations using Dot [89] and the call graph is depicted in Figure 4.6.

All systems were numbered and are represented by nodes in the graph; the Btrieve database system is shown at the bottom. Each system had outgoing calls to the database system and to at least one other system. On average, a system had outgoing calls to four systems (excluding the calls to the database system). Five systems had outgoing calls

to at least ten systems (System 23, 26, 32, 33 and 43). Of the 45 systems, 26 systems had incoming calls and six systems had incoming calls from more than ten systems (System 2, 26, 32, 33, 36 and 43). One system had incoming calls from all 44 other systems (System 2).

The relations can be further refined by partitioning it into clusters using Relation Partition Algebra [124]. To gain more insights into the dependencies, we calculated the transitive closure of the calls. Hence, if System 1 calls System 2, and System 2 calls System 3, System 1 transitively calls System 3. Note that, as we abstracted from the exact programs that performed the calls, this calculation may yield false positives. A more precise analysis, including the internal calls and dynamic calls, can be used to increase the accuracy of the outcome. With the calculated transitive closure of the calls, it turned out that there was a core set of 22 systems that had a transitive call relation with each other. This implies that it is difficult to test one of these systems alone, as it depends on all of the other 21 systems. We depicted the partitioned call graph in Figure 4.7. The figure shows the original relations plus the transitive call relations of the 22 interdependent systems. At the top, there are 19 systems that have no incoming calls and 4 systems with one incoming call, originating from one of the 19 systems without incoming calls. All systems at the top have a transitive outgoing call to each of the interdependent 22 systems, but this is not shown in the graph.

The call graph was constructed using a lightweight lexical approach, but we quickly gained insights into the relations among the systems. The dependencies in the figure illustrate that many of the systems are strongly interrelated with each other, and that modifications to a system in the portfolio cannot be tested properly until all interdependent systems have been updated as well.

4.2.4 Why a manual approach is infeasible

A simple code analysis for estimating the amount of code to be changed was done with a one line `grep` command, reporting 3,478 relevant calls spread over 1,930 files. It is tempting to think that one can apply a brute force approach to inspect and modify these calls. If we had taken 5 minutes per call, this would have resulted in 290 hours of effort (5 minutes $*$ 3,478 calls). Within two weeks, a team of 4 programmers could have done the work. At first sight, this appears to be a reasonable solution. But portfolio-wide mass changes are beyond the scope of the day-to-day routine of normal maintenance, and such changes cannot be carried out quickly and accurately by a team of programmers. We explain that here.

It is feasible to change one line of code in one file within 5 minutes, i.e. open a file, search the relevant line, change it, search the rest of the file, and close the file. Then re-compile the program and test it. Perhaps two lines of code in two files can be done in 10 minutes, including compilation and testing. But 3,478 changes in 1,930 files in 45 systems cannot be done in 290 hours, because portfolio-wide mass changes do not scale linearly.

First of all, it is time-consuming to compile and test a large amount of code, so it is infeasible to do this after every single change. One can choose to compile and test after a number of changes, for instance, first modify the programs of a single system. How-

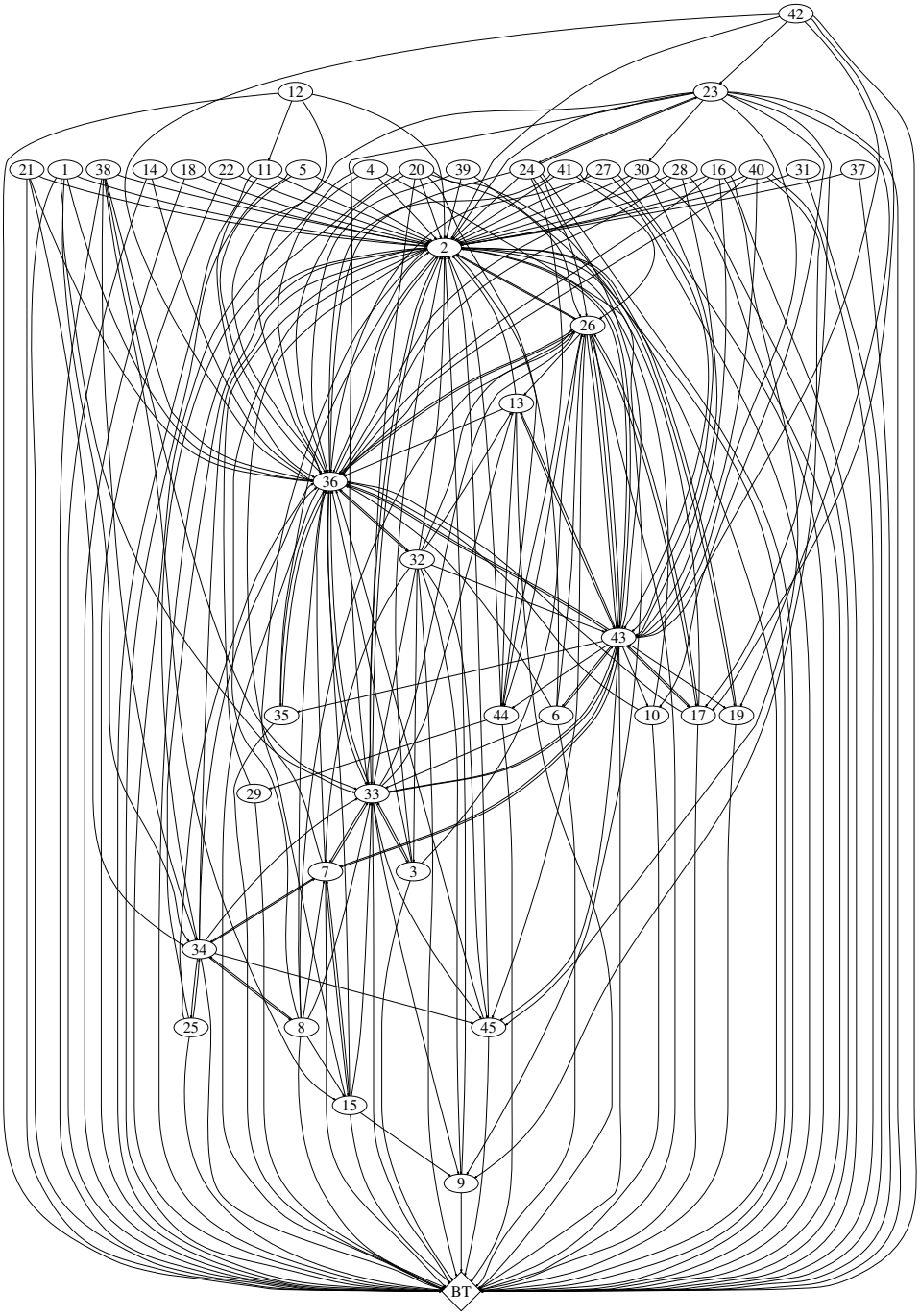

Figure 4.6: Call dependencies among the systems in the portfolio. The database system is shown in the diamond at the bottom of the figure.

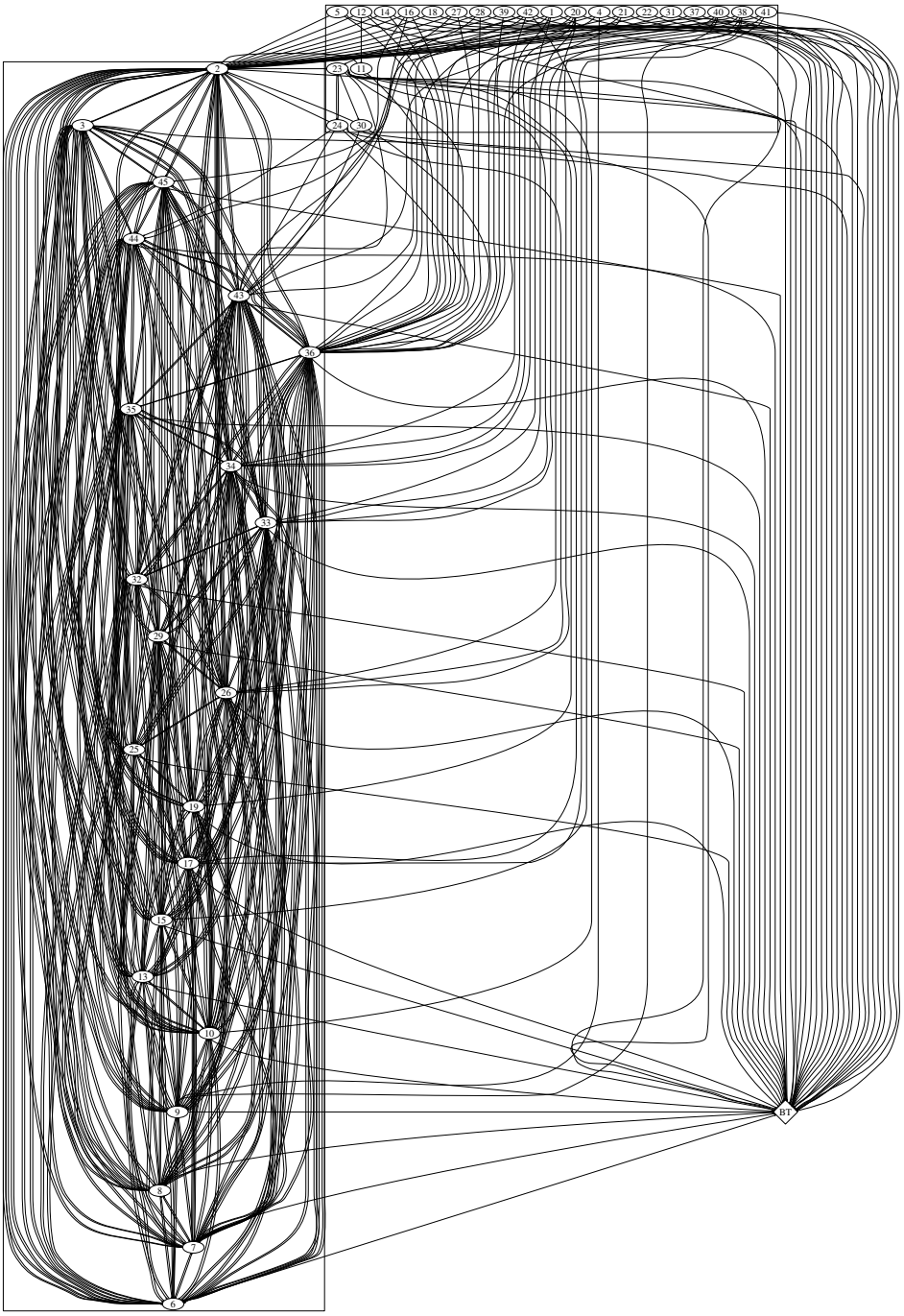

Figure 4.7: Partitioned call graph of the portfolio with the transitive call relation of 22 systems on the left-hand side. At the top, 19 systems have no incoming calls.

ever, several systems in a portfolio are often intertwined with each other, as we illustrated earlier. A partial update will therefore cause problems. For example, when a program in the updated system makes a call to a program that has not yet been updated, an error can occur. Then, the origin of the error is unclear; it can be caused by a compatibility problem or by an erroneous update. In our case, the new database version imposed several constraints on the programs, and it did not work properly with programs that have not yet been updated. This means that updated programs cannot be tested unless all dependent programs are updated as well. To accurately determine all dependent programs is already a difficult analysis in itself (e.g. analysis of dynamic calls among different systems), and in practice an infeasible task by hand.

Furthermore, as long as not all programs are updated, it is not convenient to perform other maintenance tasks since these can interfere with the upgrade project. To avoid difficult merges, the state of the portfolio must be fixed during the mass change, which hinders regular maintenance activities. In addition, manual modifications are prone to errors, even if it concerns one-line changes. Recall that it is reported that 55% of all one-line maintenance changes is erroneous [86, p 333].

Finally, mass changes have to be carried out several times during a project for various reasons. In the Btrieve project, we applied our automatic tools 6 times: one time because some code was missing, one time on a few programs, one time on one system, one time because the code had changed since the start of the project, one time to correct errors, and another time because the requirements had changed. We even had to undo one of the requested changes, which is not recommended to do by hand. In fact, in an earlier attempt, the company had tried to carry out some of the changes by hand. The attempt failed, but the changes that had already been made were not reverted. This complicated future changes and led to the cancellation of one of the requested changes near the end of the Btrieve project.

4.3 Automated solution

Here, we present our approach to carry out the changes to the software portfolio. We describe the different phases in the update process, elaborate on the used technology, and discuss the implementation of the tools.

The size and the nature of the problem make it very suitable for an automated solution. To do automatic analyses and transformations, we developed a mass update factory that used a mixture of lexical and syntactic tools. Lexical tools are particularly useful for quick lightweight analyses as we showed above, and some types of modifications. Syntactic tools are useful for more accurate and in-depth analyses and modifications. In Section 4.3.2, we briefly discuss the pros and cons of lexical and syntactic tools.

4.3.1 Mass update factory

A mass update factory is an instance of a software maintenance factory, and has the architecture we described in Chapter 2 and depicted in Figure 2.12. We briefly reconsider its structure and tools. Source code is processed in the mass update factory on a conveyor

belt with several phases, containing tools that can be reused or generated quickly to deal with specific problems:

- Preprocessing
- Parsing
- Transformation
- Unparsing
- Postprocessing

Preprocessing The preprocessing phase comprises several activities. A preprocessor massages the code to ease the parsing and the transformations, i.e. the number of syntactic possibilities is reduced without changing the behaviour of the code. For example, comments are encapsulated such that they are preserved and can be retrieved later, keywords are changed to uppercase, and include files are included. A Cobol preprocessor for software maintenance purposes is described in [41].

Parsing After preprocessing the programs, the next phase is parsing. Parsing can be done by a syntactic analysis of the source code using a grammar of the programming language, thereby constructing a parse tree. This tree is then used to apply transformations. A parser can be generated using the ASF+SDF Meta-Environment [44].

Transformation After the programs are parsed, the actual modification can be applied. Our transformations were implemented in the ASF+SDF Meta-Environment, which supports conditional rewrite rules. This way, modifications can be specified using intuitive code patterns that are applied under certain conditions.

Unparsing Unparsing is the process of translating the (transformed) parse tree back to text [47]. Unparser functionality is supplied by the ASF+SDF Meta-Environment.

Postprocessing The postprocessing phase is mainly the inverse of the preprocessing phase. For example, the comments are turned back into their original state and the expanded copybooks are collapsed.

Several of the tools that were used in the factory were generated from a context-free grammar, as described in [44]. This way, we were able to deploy tools quickly. The generated tools were then placed in a software maintenance factory architecture [173] (see also Chapter 2). In the next sections we describe the used technology and more implementation details.

4.3.2 Technology

In Chapter 2, we discussed tools and technology for large-scale software transformations. Here, we briefly reconsider the capabilities and limitations of these technologies, and how we deployed these technologies to carry out the automated maintenance.

The automated modifications were carried out using a combination of lexical and syntactic technology. Lexical technology operates at the character level whereas syntactic technology operates on the language's syntax level.

Lexical and syntactic technology

We showed in the previous section that with the `grep` command we quickly estimated the number of database calls and number of files that may be affected by the database upgrade. This information is gathered with very small effort using lexical tools. For somewhat more advanced analyses and also for some kinds of transformations, Perl [170] can be deployed. It is a powerful scripting language that can be used to manipulate text and files by replacing strings and regular expressions. The project described in [117] was carried out using Perl. For some types of text replacements, lexical technology can be applied.

However, when a change requires a certain amount of context information and must be carried out accurately, a lexical tool can become truly intricate. An accurate match pattern for a single line may be a difficult operation, but an accurate pattern for multiple lines requires much effort or becomes impossible with lexical tools. The chance of false positives and false negative is high due to missing context and the tools have limited flexibility. As noted in [44, p 248], the power of lexical tools for source code transformations is quite limited. As we mentioned earlier, in that paper an extensive example is discussed where one line of code must be altered. The authors elaborate on the problems with a lexical approach. Seemingly simple changes do not imply simple tools. So when accuracy and more context are required, lexical technology will just not do and one has to resort to syntactic technology. Syntactic tools are grammar-based so these are aware of a language's syntax, which has several advantages over lexical tools. Since a program is parsed entirely before making changes, one can implement accurate patterns that use the entire program as context. Sophisticated analyses and modifications can then quickly be developed. Nevertheless, syntactic technology has some challenges: the upfront investment in a grammar, the overhead to apply the technology, and the robustness of a parser. Furthermore, syntax retention has to be considered, i.e. it is often required that code that is not affected by an update should remain unchanged. In practice, this means that original layout and comments must be preserved by an automatic tool. In the Btrieve project, we opted for a mixture of lexical and syntactic tools to combine the best of both technologies.

The ASF+SDF Meta-Environment

To specify and implement our transformations we used the ASF+SDF Meta-Environment [45, 48, 112], which we discussed in detail in Chapter 2. We briefly review the abilities of the ASF+SDF Meta-Environment for source code transformations, such that the upcoming details of the project can be explained properly.

```
%% SDF production rule
  Non-terminal1 "Terminal" Non-terminal2
                       -> Resulting-non-terminal {attributes}

%% ASF rewrite rule
[asf-rewrite-rule-tag]
  Condition1 && Condition2 == true
  ===>
  Left-hand-side = Right-hand-side
```

Figure 4.8: An SDF production rule and an ASF rewrite rule.

In a transformation project, SDF can be used to specify the production rules of a grammar and syntax rules for transformations, and ASF can be used to specify rewrite rules for transformations. Figure 4.8 shows the structures of an SDF production rule and an ASF rewrite rule. An SDF production rule has zero or more (non-)terminals on the left-hand side and the resulting non-terminal on the right-hand side. It can also have attributes to guide the parsing or rewriting process. The grammar is used to generate a parser. An ASF rewrite rule has a left-hand side and a right-hand side indicating the rewrite relation with an equal sign (=), and can have zero or more conditions which are specified above the arrow (===>). The operators for conditions are equality (==), inequality (!=) and matching assignment (:=). In an ASF rewrite rule, variables can be used to represent actual syntax in the code to be transformed, which allows one to specify code patterns with abstract and concrete syntax. In addition, the ASF+SDF Meta-Environment provides very convenient support for generic traversal [35]. These *traversal* functions implicitly traverse a parse tree and are defined for specific non-terminals by the user; users do not have to implement this visitor behaviour themselves. At each visited node in the tree which is of these non-terminals, the rewrite rules for this function are tried. Traversal functions were first used with the ASF+SDF Meta-Environment in [40, 44] to reduce the manual effort to write renovation transformations. This turned out to be a powerful asset, and was incorporated in the ASF+SDF Meta-Environment. In the Btrieve project, we employed traversal functionality extensively to implement rewrite rules concisely.

The ASF+SDF Meta-Environment comes with a scannerless Generalised LR parser [206]. We mention that GLR parsing is particularly helpful for reengineering and maintenance, as discussed in [43]. It is also mentioned in [24] that the traditional parsing techniques are not suitable for automated software maintenance. Furthermore, the ASF+SDF Meta-Environment provides an unparser to translate parsed code to text, as well as support for pretty printing. See [38, 47] for details.

We already mentioned that a common challenge in automatic modification of software is syntax retention. In the Btrieve project, we needed to preserve the original comments. The ASF+SDF Meta-Environment supports rewriting with layout [46], which means that comments encapsulated in layout are preserved, except in places where code is rewritten. For the Btrieve project, this was sufficient because the changes were applied locally, so no comments were lost. We also added comments to places where code was modified.

```
"CALL" Identifier-literal
  ( "USING" (
      ( "BY"? "REFERENCE"? ( Identifier-filename
                           | ("ADDRESS" "OF" Identifier)
                           )+ )
    | ( "BY"? "CONTENT"    ( (("LENGTH" "OF")? Identifier)
                           | ("ADDRESS" "OF" Identifier)
                           | Literal
                           )+ )
    ) +
  )?          -> Call-statement-simple
```

Figure 4.9: SDF production rule for the IBM VS Cobol II `CALL` statement.

Grammar engineering

Before the source code can be parsed, a grammar is needed which describes the syntax of the source code language. Our starting point was an IBM VS Cobol II grammar in SDF which was derived from the online IBM VS Cobol II grammar [129, 130, 131]. We adapted the grammar such that we were able to parse Micro Focus Cobol. We added new productions and we modified existing productions. Figure 4.9 shows the SDF production for the `CALL` statement in IBM VS Cobol II and Figure 4.10 the adapted production for Micro Focus Cobol. Micro Focus Cobol supports the ”call by value” extension so we adapted the IBM VS Cobol II production rule accordingly. Also, the Btrieve database call utility, ”BT”, was added as an optional terminal denoting the BT command for the database interface. We added several other production rules for both lexical and context-free syntax.

Due to time constraints, we adapted the grammar by hand in an ad hoc way to suit the projects requirements, but a formal approach to grammar adaptation is described in [126] and tool support is given by the Grammar Deployment Kit (GDK) [122], which can be used for automatic modification of grammars. We discussed GDK in Chapter 2. An elaborate agenda on grammarware is given in [113]. Grammarware comprises grammars and all grammar-dependent software.

4.3.3 Implementation of the tools

We deployed a mixture of lexical and syntactic tools in our factory to analyse and update the portfolio. Our preprocessor and postprocessor were implemented in Unix shell scripts and Perl, our transformation tools in the ASF+SDF Meta-Environment. The cursor analysis tool was also implemented in Perl. The actual implementation of the factory's conveyer belt was done by connecting the individual tools using Unix shell scripts.

Although some of the tools might give the reader the first impression that this is an all too specific approach to solving mass maintenance problems, we like to stress that this is not the case. Since many mass maintenance problems are indeed idiosyncratic, you should not build a technology based on such examples. Instead, we developed a software product line for mass maintenance tools which, given some specific problem, is capable of

```
"CALL" "BT"? Identifier-literal
  ( "USING" (
      ( "BY"? "REFERENCE"? ( Identifier-filename
                           | ( "ADDRESS" "OF" Identifier )
                           | Literal
                           )+ )
    | ( "BY"? "CONTENT"    ( (( "LENGTH" "OF")? Identifier )
                           | ( "ADDRESS" "OF" Identifier )
                           | Literal
                           )+ )
    | ( "BY"? "VALUE" Identifier-literal )+
    ) +
  )?          -> Call-statement-simple
```

Figure 4.10: Adapted SDF production rule for the CALL statement in Micro Focus Cobol.

```
getinput | $expandcopy -sys "$system" -name "$infile"
  -src "$srcpath" -top "$toppath" -cpy "$cpypath" |
  $uppercase | $encapcomment | dumpoutput
```

Figure 4.11: Preprocessor coordination pipeline, implemented with Unix pipes.

quickly implementing a tool solving that problem without putting a lot of problem specific effort in the tool itself. The grammar of the language is the basis of this approach, and is used to generate tools for the problem at hand. For instance, a parser and generic rewrite functionality can be generated that can quickly be adjusted to a project's needs.

Factory components

preprocessor and postprocessor In Figure 4.11 the coordination script of the preprocessor is shown, implemented using Unix pipes. The postprocessor has similar functionality as the preprocessor, but then in a reversed order so we will not discuss the postprocessor in detail. We will explain the stages of the preprocessor here; we mention that the bottom-line here is that each program which went through the update process but was not altered by the transformations must remain unchanged, so the postprocessor has the task of assuring this property.

The reuse of a generic component such as a preprocessor is one of the advantages of a software renovation factory. First, we expand the copybooks, which means that we insert the actual text of the copybook in the program, similar to what a Cobol preprocessor does. We used a Cobol preprocessor for software maintenance purposes which is described in [41] and adjusted it to the needs of the Btrieve project. In Cobol, variables can be substituted by copy replacings when files are included; this could complicate the expansion. However, we checked in advance that this did not occur in the portfolio with a simple `grep` command. If it did occur, copy replacings should have been handled by the preprocessor. Expansion of the copybooks simplified the transformations because after expansion all variables are declared inside the program; a different option is to search

```
%% case-sensitive keywords
  "IF" Condition "THEN"?
      Statement-list
  Else-phrase?                        -> If-statement-non-closed

%% case-insensitive keywords
  If-KW Condition Then-KW?
      Statement-list
  Else-phrase?                        -> If-statement-non-closed

  [Ii][Ff]                            -> If-KW
  [Tt][Hh][Ee][Nn]                    -> Then-KW
```

Figure 4.12: Case-sensitive and case-insensitive Cobol `IF` statement specified in SDF production rules. The case-insensitive variant requires additional production rules for the keywords.

the copybooks during the transformation. We chose to do expansion in the preprocessing because that would be faster than to let a transformation search for the copybooks each time we must check if a variable is declared or what its value is. With our approach, the transformation rules do not have to take copy statements into account since all data declarations have been included in the program. The expansion requires the system name, the file name, and three paths for searching the copybooks. Since there were several versions of the same copybooks in the delivered source, the appropriate copybook must be found using the correct precedence of the different paths where a copybook could be located. This was done in consultation with the system experts. The original copy statement remains in the program, and two dummy copy statements indicate the start and end of the copybook (`COPY EXPAND-START` and `COPY EXPAND-END`). These are cleaned up together with the content by the postprocessor. Note that we do not have to change the Cobol grammar for the dummy copy statements, because they are perfectly legal in Cobol. Since the syntax of the language remains the same, this way of temporarily including information in a program is called *native scaffolding*. Scaffolding [174] is related to a very common concept in the development of software systems: the use of code, data, or entire programs which are built for debugging or tracing purposes, but never intended to be in the final product. We temporarily included and marked the copybooks in the programs by using the syntax of the Cobol copy statement; hence, we did not have to modify the syntax.

The next step is to convert the lowercase characters of keywords and special names to uppercase characters. Since the ASF+SDF Meta-Environment does not support case-insensitivity, mixed case terminals have to be specified in the grammar. We illustrate this in Figure 4.12, where a case-sensitive and a case-insensitive Cobol `IF` statement is shown. A case-insensitive SDF grammar production requires additional productions for the terminals. This complicates the grammar [41] and imposes performance constraints on the parser which is supplied with the ASF+SDF Meta-Environment. So, instead of making the grammar case-sensitive, we opted for changing all keywords to uppercase.

The final step is to encapsulate all Cobol comments. In Cobol, comments are indicated

```
  if ($line =~ /^([\ ]{0,6})\*/) {
    if ($1 =~ /[\t]/) {
    } else {
      $line = "\%\%".$line;
    }
  }
```

Figure 4.13: Perl code for encapsulating a Cobol comment with two percentage signs using the preprocessor.

```
        ...
        copy h:\rlo\claim\teb415.rlo.
        ...

        FILL-NUMBER SECTION.
        fn01.
                if sw-debt = 1 or 2
                        go to fn99.

*               if not lw-number = zero
*                       go to fn20.
```

```
        ...
        COPY h:\rlo\claim\teb415.rlo.
        COPY EXPAND-START.
        77 teb415-position PIC X(37).
        77 ec-teb415 PIC X(128).
        77 dbl-teb415 PIC 99 COMP-0 VALUE 12.
        01 buf-teb415 PIC 9(6).
        01 teb415-record.
           03 teb415-nr-claim PIC 9(6).
           03 teb415-nr-pts PIC 9(6).
        COPY EXPAND-END.
        ...

        FILL-NUMBER SECTION.
        fn01.
                IF sw-debt = 1 OR 2
                        GO TO fn99.

%% *              if not lw-number = zero
%% *                       go to fn20.
```

Figure 4.14: A Cobol code snippet before and after preprocessing.

```
[2b]
  equal( Data-name1, Data-name2 ) == true
  ===>
  analyse-variable(
    77 Data-name2 Data-description-entry-clause*.
  , Data-name1 )
  =  true
   ; calc-length(77 Data-name2 Data-description-entry-clause*.)
   ; find-value(Data-description-entry-clause*)
```

Figure 4.15: An ASF rewrite rule for matching a level 77 Cobol variable and retrieving its length and its value.

by a comment indicator (∗ or /) in column 1 through 7 in the source code. We set the comments aside by inserting them into the layout; this way, we do not have to take care of the comments everywhere they can appear in the Cobol grammar. We encapsulate comments with percentage signs (%%), which is the standard comment notation in SDF. In Figure 4.13, a Perl code snippet is shown for matching the comment indicators in columns 1 through 7 and inserting the percentage signs; note that Perl starts counting at 0 for the first column and that lines starting with a tab are skipped.

In Figure 4.14, we show an original code snippet before and after preprocessing. Keywords are capitalised, the copybook `teb415.rlo` is expanded, and comments are encapsulated.

transformation tools We implemented three modification tools using the ASF+SDF Meta-Environment, one for each requested modification. In addition, we implemented an analysis tool for finding a variable, its length and value, and a tool which adds a data declaration if it does not exist. These additional tools were used by the three modification tools.

Part of the variable analysis tool is shown in Figure 4.15. The figure shows an ASF rewrite rule which matches and analyses a level 77 data declaration, which is a regular variable declaration in Cobol. If the searched variable `Data-name1` is equal to the current variable `Data-name2`, an argument is returned that is composed of three values, separated by semicolons: `true` to indicate a successful match, the length that is calculated by `calc-length`, and the initial value that is retrieved by `find-value`. This analysis has several other rules for matching other types of data declarations, i.e. other level numbers, which are not shown here. In Figure 4.16, this transformation is applied to actual code for analysing a `KEY-0` variable. The function is applied to a list of data declarations, and `KEY-0` is supplied as the variable we are looking for. If the `KEY-0` variable is found, `true` is returned together with length `2` and value `0`. If no matching variable is found, a default rule is applied. This analysis is used by the other transformation rules. For the Key-0 and Dbl-4 modification, we need to know whether the `key-0` and `dbl-4` variables have already been declared, and for the buffer-length modification we calculate the length of the buffer itself and retrieve the value of the buffer-length variable.

```
Input term
analyse-variable(
77 B-CODE               PIC X(8).
77 KEY-0                PIC 99 COMP-0 VALUE 0.
77 KEY-1                PIC 99 COMP-0 VALUE 1.
77 KEY-2                PIC 99 COMP-0 VALUE 2.
77 KEY-3                PIC 99 COMP-0 VALUE 3.
, KEY-0
)
Output of the transformation
  true ; 2 ; 0
```

Figure 4.16: An example of `analyse-variable` from Figure 4.15 applied to a Cobol code fragment. The transformation rule is applied to data declarations to search and analyse the `KEY-0` variable. At the bottom of the figure, the result of the transformation is shown: the variable is found, its length is 2 and its value is 0.

In Figure 4.17, we show three rules from the Key-0 modification. We briefly explain the rules.

- `[eq-key-0]` the function `key-0` matches a Cobol program, which is composed of several divisions. The database calls are located in the procedure division. The function `key-0-modify` searches the procedure division for database calls with relevant operations. If at least one modification has been made (`Procedure-division'` is different from `Procedure-division`, which is checked by the `equal` function), the function `add-data-entry` is called to add a `KEY-0` declaration to the data division. This function adds a variable if it has not yet been declared. The updated procedure division `Proceduredivision'` and the possibly modified data division `Data-division'` then replace the original divisions on the right-hand side of the rewrite rule.

- `[eq-key-0-modify]` the function `key-0-modify` matches database calls with the `B-RES` (reset) and `B-UNL` (unlock) operation. If the key number is not equal to `KEY-0`, the `CALL` statement is modified. This function is a traversal function, which means that it visits all `CALL` statements implicitly; we do not have to implement a visitor function ourselves. At each visited `CALL` statement, the rewrite rule is tried. If the rule matches, a special comment code `@!KEY-0!@` is added to the statement, indicating that it has been modified.

- `[eq-add-data-entry]` the function `add-data-entry` matches a data division, which consists itself of several sections. In each of these sections, there can be declarations. All declarations are collected and then `analyse-variable` checks if the supplied `Data-name` is already declared in one of these sections. If `false` is returned by `analyse-variable`, then a new variable is added

to the working storage section in the data division, together with a special comment code `@!ADD-DECL!@`. The `Integer` and `Literal` which are returned by `analyse-variable` are not used by this transformation.

In two of the rules, a special comment code is added to the transformed parts of the code to automatically document the change. These comment codes are scaffolded comments [174], which we encapsulated in layout with an at sign and an exclamation mark (`@!`); this way, we do not have to modify the grammar. The scaffolded comment is placed inside the statement for technical reasons, since outside the statement it can be lost in a rewrite step (it is considered as layout, which can be lost during rewriting in some cases), and they are changed to a Cobol comment line by the postprocessor. This way, the changes are documented in a uniform way. Furthermore, we used these comment lines afterwards to trace and measure the number of changes made by the automatic tools.

The three rules show the core of the specification for the Key-0 modification. The Dbl-4 modification is very similar except for different operations and variables, but the Data-length modification is different since we must also add a new statement. Figure 4.18 depicts some ASF code from the Data-length modification: the rule which adds the `MOVE` statement with the calculated length of the data buffer. The function `data-length` traverses a program until a list of statements with a relevant `CALL` statement is found. The initial values of variables (`Initial-values`) and calculated lengths of variables (`Lengths`) are determined in advance, as well as the variables that are modified somewhere in the program (`Modified-datanames`). If the database operation is equal to `B-STA` or `B-CRE`, and the initial value of the `Data-length` variable is smaller than the calculated length of the `Data-buf` or the `Data-length` variable is modified somewhere in the program, a `MOVE` statement is added on the right-hand side of the rule. The calculated length is moved to the `Data-length` variable; a conversion function (`int-to-literal`) converts the calculated length to a literal, which is required for a `MOVE` statement. The scaffolded comment `@!DATA-LENGTH-MOVE!@` indicates that a Data-length modification has been made.

Cursor analysis tool To carry out the cursor analysis, we initially implemented an analysis tool in ASF+SDF. We needed to search for loops in which a file was accessed and another program was called that accessed the same file. However, in consultation with the system experts, it turned out that we could simplify the analysis. We agreed that is was sufficient to search just for sections instead of loops since this turned out to be sufficient to detect the problems of our customer. For this simplified analysis, we chose to implement two Perl scripts instead of an ASF+SDF analysis tool. If a complete control-flow analysis was required to detect the problems, then we would have used syntactic tools instead, as that is quite a complicated task to perform using lexical tools. For example, in the code sample in Figure 4.4, various `GO TO` statements are employed to express the logic. A control-flow analysis of such code with a lexical tool becomes very intricate. To analyse such code properly, a goto-removal transformation is helpful to normalise the control-flow [42, 171, 194]. However, that would change the focus of the project.

The first script extracted for each program the section names with called programs, each section with database operations that can alter the cursor position with the used data

```
[eq-key-0]
  Procedure-division' := key-0-modify( Procedure-division )
  equal(Procedure-division',Procedure-division) == false
  Data-division' := add-data-entry( Data-division
                    , 77 KEY-0 PIC 99 COMP-0 VALUE 0. )
  ===>
  key-0( Identification-division
         Environment-division
         Data-division
         Procedure-division )
  = Identification-division
    Environment-division
    Data-division'
    Procedure-division'

[eq-key-0-modify]
  equal(Operation,B-RES) | equal(Operation,B-UNL) == true
  equal(Key-number,KEY-0) == false
  ===>
  key-0-modify(
  CALL BT "__BTRV" USING Operation B-status Pos-block
       Data-buf Data-length Key-buffer Key-number )
  =
  CALL @!KEY-0!@ BT "__BTRV" USING Operation B-status Pos-block
                           Data-buf Data-length Key-buffer KEY-0

[eq-add-data-entry]
  false ; Integer ; Literal :=
  analyse-variable(
    select-file-data-entries(File-and-sort-description-entry*)
    Data-description-entry*1
    Data-description-entry*2
    select-screen-data-entries(Screen-description-entry*)
  , Data-name)
  ===>
  add-data-entry(
    DATA DIVISION.
      FILE SECTION. File-and-sort-description-entry*
      WORKING-STORAGE SECTION.Data-description-entry*1
      LINKAGE SECTION. Data-description-entry*2
      SCREEN SECTION. Screen-description-entry*
   , Level-number Data-name Data-description-entry-clause*. )
  = DATA DIVISION.
      FILE SECTION. File-and-sort-description-entry*
      WORKING-STORAGE SECTION. Data-description-entry*1
      Level-number @!ADD-DECL!@ Data-name
        Data-description-entry-clause*.
      LINKAGE SECTION. Data-description-entry*2
      SCREEN SECTION. Screen-description-entry*
```

Figure 4.17: Three of the ASF rules for the Key-0 modification.

```
[eq-data-length]
  equal(Operation,B-STA) | equal(Operation,B-CRE) == true
  get-int(Data-length,Initial-values)
    < get-int(Data-buf,Lengths)
  | is-in(Data-len,Modified-datanames) == true
  Length-literal := int-to-literal( get-int(Data-buf,Lengths) )
  ===>
  data-length(
    Statement*1
    CALL BT "__BTRV" USING Operation B-status Pos-block
         Data-buf Data-length Key-buffer Key-number
    Statement*2
  , Modified-datanames
  , Initial-values
  , Lengths )
  =
  data-length( Statement*1, Modified-datanames, Initial-values
             , Lengths )
  MOVE @!DATA-LENGTH-MOVE!@ Length-literal TO Data-length
  CALL BT "__BTRV" USING Operation B-status Pos-block
            Data-buf Data-length Key-buffer Key-number
  data-length( Statement*2, Modified-datanames, Initial-values
             , Lengths )
```

Figure 4.18: An ASF rewrite rule for the Data-length modification.

buffer, and also the sections that were performed from another section. The performed sections were necessary to do a control-flow analysis in order to track down all called programs in a section. There were 30 database operations that changed the cursor position in a file. In Figure 4.19, we show the analysis of the programs from Figure 4.4.

The second script combined the data from the first script and reported for each relevant program the sections, the program that was called and also the record with the modified cursor. This was done by propagating the called program from section to section. The result of the second script is shown in Figure 4.20. With the results of the second script, the system experts made the changes to save the cursor before the calls and restore them after the calls, as we showed earlier in Figure 4.5.

Changing requirements, changing tools

While we were implementing and testing our tools, several questions arose. This happened when we examined one of the releases of the modified code and discussed our findings with the system experts. We summarise some issues here.

The Key-0 modification had to be carried out in order to make sure the key number variable in the reset and unlock operations is `key-0` with value 0. Our initial implementation replaced all key number variables that were not `key-0` and also declared this variable if necessary. However, in many programs we examined there was already a different key number variable that also had value 0 (about 326 of the 2225 reset and unlock operations). We consulted the system experts about this issue, and they decided that the requirements should change. This meant that we did not have to change the reset and unlock operations where the key number was variable `acc-0`, since its value was also 0. Our tools were quickly updated to implement the changed requirements by modifying one of the rules we showed earlier. The modified rule is shown in Figure 4.21; a simple change is made in the condition where the key number is now also tested for `acc-0`. After that, an updated tool was quickly generated and the update factory was run again. This change in our tools resulted in 326 `key-0` changes less and added one `key-0` data declaration less.

Another change to the initial requirements was caused by differing opinions about the physical length of variables. In Cobol there are several storage types for numeric data items. The reasons for this are speed of calculations as well as memory usage; depending on the Cobol dialect different types are supported. For instance, a data item can be stored as type binary which means that more than one digit is stored in one byte. Some other types are display, computational, packed decimal, and pointer, which are suitable for specific uses. The default type is display. The computational type itself is also divided into different types such as single and double precision floating point. The precise amount of bytes used for the types and the order in which they are stored heavily depends on the storage mode (byte or word), the operating system, and the compiler flags.

For the Data-length transformation, we created a function to compute the physical length of a variable, i.e. the number of bytes it takes. We needed this value to update the database call with the correct length of the data buffer, as we showed earlier. Since the physical lengths depend on several things, we consulted the system experts for the precise amount of bytes used by each storage type. The types that appeared in the portfolio were: display, computational-0, computational-3, computational-4, computational-5 and

```
PROGRAM1                                    # program name
;
...
GET-PART-AV:PROGRAM2                        # called programs
...
;
...
SEARCH-POLICYHOLDER:b-gge:file001-record # relevant operations
SEARCH-POLICYHOLDER:b-gne:file001-record # per section
...
;
...
SEARCH-POLICYHOLDER:FILL-TABLE              # performed sections
...
FILL-TABLE:GET-PART-AV
...
;
PROGRAM2                                    # program name
;
...                                         # relevant operations
DETERMINE-POLICY-RIGHT:b-gle:file001-record # per section
DETERMINE-POLICY-RIGHT:b-gpr:file001-record
...
;
...
```

Figure 4.19: Part of the analysis result of the code from Figure 4.4.

```
System 34, topprogram PROGRAM1
 Section :"SEARCH-POLICYHOLDER", record:"file001-record",
cursor reset:"PROGRAM2". Via GET-PART-AV.
```

Figure 4.20: Result of applying the second analysis script on the analysis result from Figure 4.19.

```
[eq-key-0-modify]
  equal(Operation,B-RES) | equal(Operation,B-UNL) == true
  equal(Key-number,KEY-0)
  & equal(Key-number,ACC-0) == false    %% added for ACC-0
  ===>
  key-0-modify(
  CALL BT "__BTRV" USING Operation B-status Pos-block
       Data-buf Data-length Key-buffer Key-number )
  =
  CALL @!KEY-0!@ BT "__BTRV" USING Operation B-status Pos-block
                          Data-buf Data-length Key-buffer KEY-0
```

Figure 4.21: Updated ASF rule for Key-0 modification.

Table 4.2: Physical length of the storage types in the portfolio.

Type	Length according to the system experts
`display`	each digit occupies one byte (default)
`comp-0`	up to five digits occupy two bytes, more than five digits see the display type
`comp-3`	each two digits is one byte plus a half byte for the sign
`comp-4`	same as comp-0
`comp-5`	same as comp-3
`comp-x`	each two digits occupy one byte

```
77 dbl-stat                      pic 999 comp-0 value 250.

01 status-block.
   03 etq-rl                     pic 99 comp-0.
   03 etq-ps                     pic 9(4) comp-0.
   03 etq-rsv                    pic 9(4) comp-0.
   03 key-specs occurs 20 times.
      05 key-position            pic 9(2) comp-0.
      05 key-length              pic 9(2) comp-0.
      05 key-flag                pic 9(2) comp-0.
      05 filler                  pic x(10).
...

LOG-START SECTION.
ls01.
        move 300 to dbl-stat.
        call BT "__BTRV" using b-sta, b-stat, fb-tet212,
                     status-block, dbl-stat, buf-64, key-0.
...
```

Figure 4.22: In an earlier attempt to solve the Data-length problem, a `move` statement was added to guarantee the size of length variable.

computational-x. Initially the system experts agreed upon the lengths given in Table 4.2; `comp` is an abbreviation for computational.

However, when we examined the database calls that needed to be modified for the Data-length modification (status and create operations), in many cases a `move` statement had already been added to change the length of the data buffer length variable before the call was made. This indicated that in a previous update an attempt was made to guarantee that the length variable was large enough. See the code example in Figure 4.22, where the data buffer is represented by the record structure `status-block` and the data buffer length is the variable `dbl-stat`. The initial value of `dbl-stat` is 250. According to Table 4.2, the physical length of `status-block` is:

$$2 + 2 + 2 + 20 * (2 + 2 + 2 + 10) = 326$$

```
  LOG-START SECTION.
  ls01.
          move 300 to dbl-stat.
* Btrieve 7.9 modification: data-length assignment.
          move 326 to dbl-stat
          call BT "__BTRV" using b-sta, b-stat, fb-tet212,
                        status-block, dbl-stat, buf-64, key-0.
```

Figure 4.23: The `move` statement from Figure 4.22 was overruled by our tools.

Our transformation tool modified the code as shown in Figure 4.23, where the initial `move` statement has been overruled.

We discussed this issue with the system experts. Among the experts, there was now a disagreement about the physical length of the storage type comp-0. Eventually, it was decided to cancel the whole Data-length modification due to lack of clarity. Since we used an automatic transformation tool to do the modification, we simply made a one-line change to our specification, re-generated the tool and re-ran the update factory. The change to the automatic tool resulted in 240 (data buffer length) changes less. The possibility to undo the change without too much effort is one of the advantages of using automatic tools to do mass maintenance.

Small-scale changes that are carried out manually can usually be reverted at low cost (e.g. with the help of a source control system). For massive changes to a large-scale business-critical portfolio, this is usually not possible and recommended. Due to the size of a portfolio, the intertwined impact of modifications and ongoing maintenance, it is difficult to properly isolate and keep track of changes. Therefore, if changes had been made manually and along the way one of them was cancelled, it would have been difficult to undo part of the changes. In the Btrieve project, this was illustrated by the failure of the previous attempt to update the database calls. The earlier (manual) attempt even complicated the Btrieve project.

4.3.4 Checking the modified portfolio

When one modifies over 4 million lines of code automatically, it is not obvious to see whether this is done as expected, i.e. the right transformation in the right place. One could try to prove that the transformations are correct in some sense. We already explained that a formal approach is not feasible in industrial projects like the Btrieve project. The mass update process consists of far more than just the actual modification itself, as we have seen. The other steps, such as pre- and postprocessing, should also be taken into account. Hence, a formal approach to finding errors in the transformation process hinders the flexibility of the automatic approach and is not cost-effective. Here, we discuss how we exercised control over the analysis and modifications in the project.

To test and check the modifications, we opted for a lightweight, practical approach that can be applied quickly to a large amount of source code. The checking process performs various lightweight (sanity) checks to detect errors in the automatic modifications. Failure indicates a possible error but success does not indicate a flawless modification. In

our opinion, this is the most suitable way to have cost-effective control over automatic modifications that have been applied to millions of lines of code. We refer to Chapter 6 for various checks that can be applied to mass maintenance transformations.

The transformations were developed and placed in a testing framework in ASF+SDF, which is part of the ASF+SDF Meta-Environment distribution [48]. Also, a debugger is incorporated in the environment. A combination of these tools allow for interactive development, (unit-)testing, and debugging of automatic transformations. We also worked on grammar-based testcase generation for transformation rules; we refer to Chapter 6 for more details.

For checking the modified source code, we used lexical tools to inspect the changes. Similar to our code exploration in the beginning of the project, such tools are very useful also in this stage. We used `grep` to search all Cobol files for database calls that were not updated accordingly, so we searched for *unexpected* changes. In the following example, we used `grep` to find all database calls with reset and unlock operations and also the next line (`grep -A 1`), which contains the key number argument in most cases. We pipe the output to another `grep` which filters the first line of the `call` statements and all lines with `key-0` and `acc-0` (`grep -v` inverts the match). In our case, the output of this query was one line of code. The line contained a key number argument that was an `acc-2` variable; it was not modified because it was a comment line. The result of this check does not guarantee that all occurrences have been replaced because a `key-0` or `acc-0` on different positions are also filtered, but we gained more confidence in our transformation tools at low cost.

```
Filtering expected changes for the Key-0 Modification:

$ grep -A 1 -e 'call BT "__BTRV" using b-res,'
            -e 'call BT "__BTRV" using b-unl,' *.CBL |
  grep -v -e 'call BT' -e 'key-0' -e 'acc-0'
TUC002.CBL:*  tet202-record, dbl-tet202, tet202-position, acc-2.
```

Another check can be made by comparing the input and output text using `diff`, which compares files line by line. The `diff` command in combination with `grep` can be used to find changes quickly. We will show an example on the next page. We used the Unix `for` command to extract all expected changes between the original and modified programs and store the result in the file `diff-input-output`. The `basename` command strips the directory name from a file. Then, we used `grep` to extract all lines that do not match any of the predicted changes `key-0` and `dbl-4`, and we also filtered the comment line that was added by our tools. All such comment lines started with '`* Btrieve 7.9`'. The output of the `diff` command also contained context information that we removed before issuing the `grep` query, but that is not shown in the example. The result of these operations is any unexpected change to the programs. In our case, the query returned no lines. Here again, an empty result does not imply a 100% correct transformation since the filtered patterns may overlap with errors but with this approach we can reduce a visual inspection of several million lines of code to a few lines, and we have more confidence in the transformation process.

```
Filtering expected differences between input and output source code:

$ for file in inputfiles/*.CBL;
  do diff $file outputfiles/`basename $file` >> diff-input-output;
  done

$ grep -v -e 'key-0' -e 'dbl-4' -e '* Btrieve 7.9' diff-input-output
```

With a simple combination of lexical tools, we were able to perform several checks quickly and at low cost, and we gained confidence in the automatic modification of millions of lines of code.

Before we applied our update tools to the entire portfolio, we did several releases with 10 programs. The system experts inspected these releases visually. One of the releases was also compiled to track down errors that were not detected earlier. When that release was approved, we prepared sources for a release of the largest system in the portfolio. After we performed several checks on that system and inspected suspicious files for errors, we sent all the modified programs back to the customer so they could compile it. However, when they compiled the changed sources, it appeared that the carriage return characters were missing in all files; these had been removed from the original sources when they were moved to our Unix-based environment, and we forgot to add these after the modification. Then the question arose why this was not detected at an earlier stage. Eventually, we found that the programs in the earlier release were examined by a system expert using a text editor, and this editor automatically added missing carriage return characters to each file. Furthermore, our lightweight checks missed the error because the checks were performed on sources from which the carriage return characters had already been removed, i.e. after they were moved to our Unix-based environment. We adjusted the postprocessor such that the carriage return character was properly added, and after that the updated portfolio was tested and accepted by the customer.

The automatic cursor analysis was tested using unittests, and during the project early analysis results were reviewed by the system experts. A manual inspection was feasible because it concerned no more than 50 programs. The first versions were too coarse-grained so there were several false positives. On the other hand, there were false negatives. It appeared that the initial set of database operations that had to be detected was incomplete. With the help of the system experts, we were able to adjust the analysis tool. When the experts and we were satisfied, we delivered a final analysis, reporting 38 programs. The report was accepted by the customer and used to make the necessary changes to the portfolio.

4.4 Results and costs

Results of our modifications A comment line documented each modification. We used these comments to track down the number and types of the changes quickly and accurately. In Table 4.3, we show the statistics of the final release: the total changes per system, the type of the changes, and the changed programs per system. The table also

Table 4.3: Detailed statistics of the modifications (final release).

System number	Programs	Lines of code	Changes	Change type[a]	Changed programs (%)
1	50	74,441	33	0 / 22 / 11	11 (22%)
2	128	28,150	24	16 / 4 / 4	14 (11%)
3	179	174,943	53	31 / 13 / 9	33 (18%)
4	32	22,295	11	4 / 5 / 2	6 (19%)
5	33	43,220	18	18 / 0 / 0	14 (42%)
6	369	365,420	321	218 / 76 / 27	215 (58%)
7	61	51,701	13	1 / 6 / 6	7 (11%)
8	38	43,649	7	3 / 2 / 2	5 (13%)
9	3	3,400	3	3 / 0 / 0	3 (100%)
10	6	8,011	14	0 / 12 / 2	2 (33%)
11	213	159,553	93	3 / 62 / 28	31 (15%)
12	21	19,881	13	13 / 0 / 0	13 (62%)
13	15	10,550	1	1 / 0 / 0	1 (7%)
14	119	70,745	62	29 / 20 / 13	40 (34%)
15	27	24,439	3	3 / 0 / 0	3 (11%)
16	2	2,515	0	0 / 0 / 0	0 (0%)
17	38	43,784	84	22 / 46 / 16	28 (74%)
18	6	2,251	3	1 / 1 / 1	1 (17%)
19	7	4,725	4	4 / 0 / 0	4 (57%)
20	102	132,599	4	4 / 0 / 0	4 (4%)
21	83	78,090	3	3 / 0 / 0	3 (4%)
22	12	8,794	6	2 / 3 / 1	2 (17%)
23	57	53,023	51	20 / 21 / 10	28 (49%)
24	22	16,648	18	6 / 8 / 4	9 (41%)
25	27	21,116	2	0 / 1 / 1	1 (4%)
26	129	175,978	173	63 / 78 / 32	70 (54%)
27	8	8,120	11	7 / 2 / 2	8 (100%)
28	25	33,755	1	1 / 0 / 0	1 (4%)
29	7	7,612	0	0 / 0 / 0	0 (0%)
30	11	9,795	2	1 / 1 / 0	2 (18%)
31	54	47,102	81	32 / 37 / 12	43 (80%)
32	3	2,232	0	0 / 0 / 0	0 (0%)
33	142	81,529	30	26 / 2 / 2	25 (18%)
34	357	451,032	158	5 / 93 / 60	70 (20%)
35	2	1,014	1	1 / 0 / 0	1 (50%)
36	138	67,387	14	14 / 0 / 0	14 (10%)
37	2	6,148	4	0 / 2 / 2	2 (33%)
38	5	4,082	0	0 / 0 / 0	0 (0%)
39	6	9,137	7	1 / 5 / 1	2 (33%)
40	38	39,134	66	2 / 47 / 17	17 (45%)
41	7	5,879	0	0 / 0 / 0	0 (0%)
42	48	45,758	43	22 / 14 / 7	24 (50%)
43	258	242,059	41	35 / 3 / 3	36 (14%)
44	40	32,057	10	10 / 0 / 0	10 (25%)
45	24	16,369	2	2 / 0 / 0	2 (8%)
Total	2,954	2,750,122	1,488	627 / 586 /275	805 (27%)

[a] Key-0 / Dbl-4 / Add-declaration

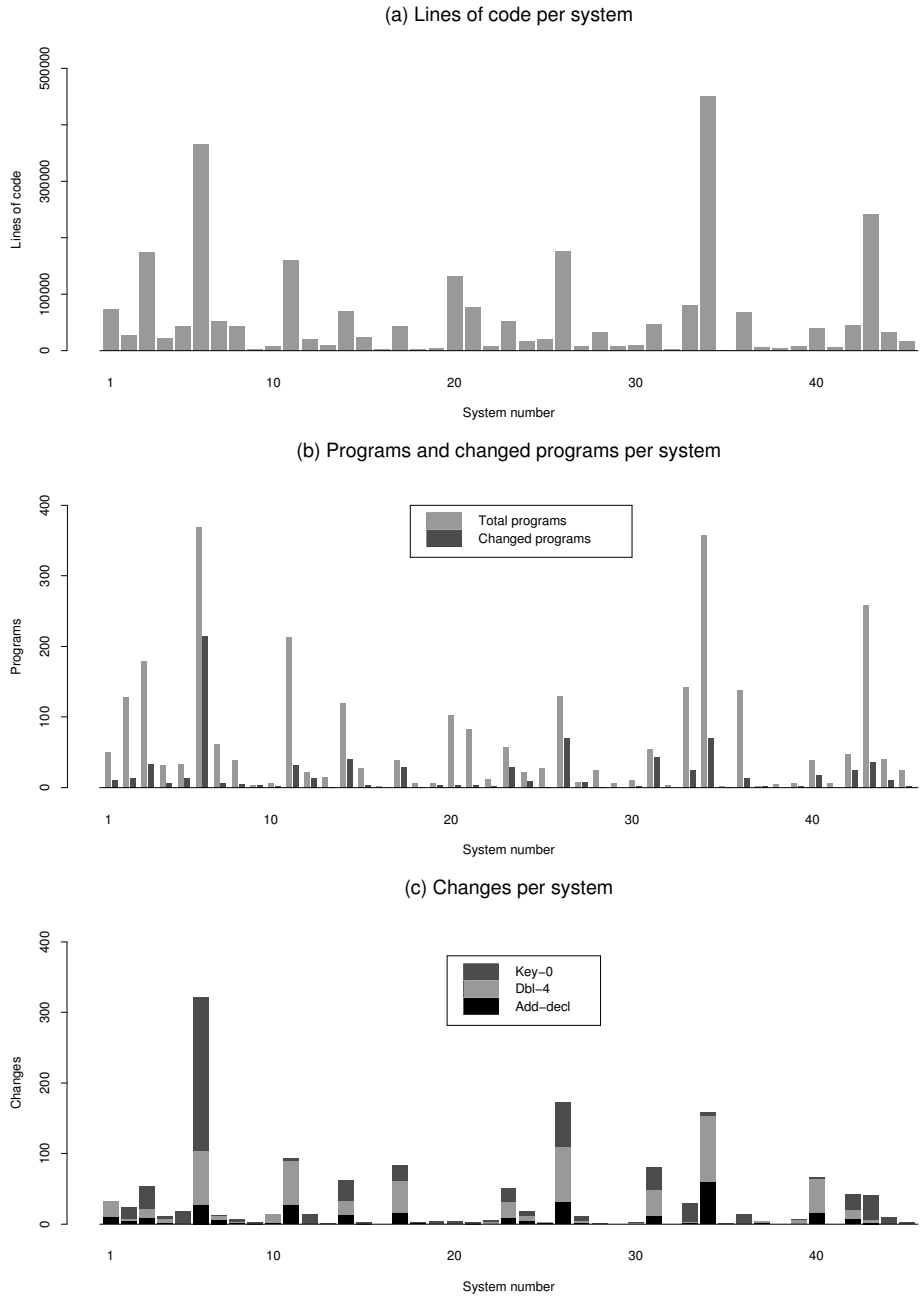

Figure 4.24: Statistics from Table 4.3 depicted.

(a) The number of lines of code for each system (without copybooks)
(b) Total number of programs per system and the number of changed programs
(c) Different types of changes made in each system

Table 4.4: Statistics of the cursor analysis.

System number	Reported programs	Reported files	Called programs
1	4	4	4
3	1	1	2
5	1	1	1
17	1	1	1
26	6	6	25
33	2	2	8
34	17	17	52
41	1	1	1
43	5	4	14
Total	38	37	108

shows that the portfolio consisted of systems with varying sizes, ranging from one thousand lines of code to nearly half a million lines of code (System 34). We have depicted the data in Figure 4.24 using S-plus [123], a tool for statistical research, data analysis and data visualisation.

Initially, the impact of the database upgrade seemed to be a small, simple change to some database calls, but we can see from the table and figure how widespread the modifications are throughout the entire portfolio. In the final release, 1,488 changes were made. Only 5 of the 45 systems were not affected by the massive update, and 805 of the 2,954 programs were changed, which is 27%. Most changes were made to System 6, where 321 changes were made to 215 of the 369 programs (58%). In two systems, 100% of the programs were changed but these were some of the smaller systems. The statistics show that the seemingly simple changes to variables in the database calls affected the entire software portfolio.

Results of the cursor analysis The results of our cursor analysis are summarised in Table 4.4. Our analysis reported 38 programs in 9 systems, which accessed 37 files with a conflicting cursor. These programs called 108 programs that caused the conflicts. The complete analysis reported the exact sections from which the programs are called, the conflicting file and the database operations that are performed. Hence, less than 40 programs needed to be altered, and with these results, the system experts manually modified the reported programs to solve the problems with the conflicting cursors.

Releases During the project, several releases were made. A release involved code that was transformed by us and returned to the customer. We did six releases in total. Figure 4.25 shows the changes that were made for each release. The first release comprised 10 programs with 26 changes in total; this release was a first test to detect simple problems. It turned out that some copybooks were missing. The second release was done with the missing copybooks added and now there were only 16 changes since 10 data declarations were added unnecessarily in the first release (these were present in the copybooks). The third release was a modification of the entire System 34, which was the largest sys-

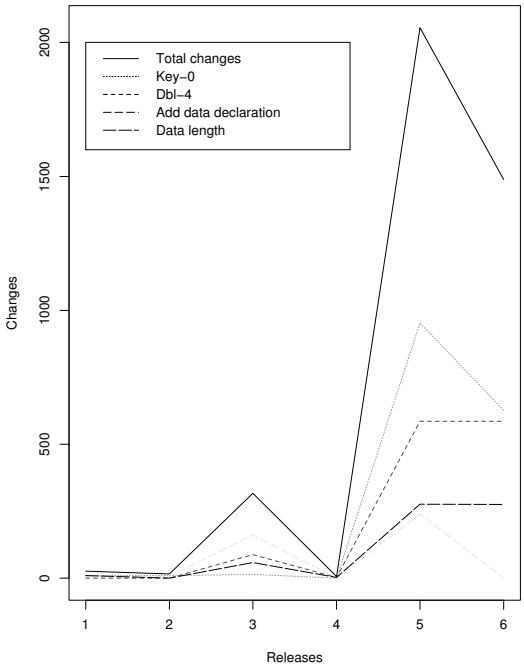

Figure 4.25: Changes made per release.

tem in the portfolio. There were 317 changes made to 450 thousand lines of code. This release uncovered the carriage return characters problem and some pretty printing issues. The fourth release consisted of three programs from System 34 to solve these problems, and thus only 6 changes were made. The fifth release was the first release with the entire software portfolio; there were 2,055 changes to 1,046 programs. After this release, we were asked to undo the Data-length modification and to add `acc-0` as a valid variable to the Key-0 modification. This resulted in 1,488 changes in the sixth release, which was the final release.

Project costs We elaborate on the costs of the projects in terms of time. We describe the effort that we spent on the project, a summary is given in Table 4.5; one day is one person day with 8 hours. Two persons worked on the project. The figures on the application are based on a Linux PC with a 600Mhz processor and 512MB memory.

Our starting point was a generic software maintenance factory, which required adjustments to this project's specific needs. For the grammar, we started with an IBM VS Cobol II grammar in SDF. We adapted this grammar to be able to parse the Micro Focus Cobol portfolio, we adjusted the pre- and postprocessor, we developed automatic transformations in the ASF+SDF Meta-Environment, and we implemented two Perl scripts for the cursor analysis. All tools were placed in an infrastructure for development, testing and batch application. Using this generic product line for software maintenance, we were able to develop everything in 32 person days.

Table 4.5: Effort spent in the project.
(1 day = 8 hours)

Task	Time
Grammar engineering	8 days
Transformation development	
- basic, reusable	8 days
- project specific	5 days
Infrastructure development	4 days
Pre-/postprocessing	2 days
Cursor analysis tool	5 days
Application	
- parsing	3 hours
- transformation	5 hours
- cursor analysis	2 hours
Communication	3 days
Total (approx.)	36 days

The work on adapting the IBM VS Cobol II grammar to the entire portfolio took 8 person days. This included extension of the existing rules, disambiguation, and testing the grammar with small sample programs. We added 119 production rules and 98 rules were modified. The adapted grammar consisted of 513 production rules. The pre- and postprocessor were reused from an earlier project [41] and adapted to the Btrieve project's specific needs, such as Micro Focus specific syntax and the copybook expansion. This took 2 person days.

The transformations consisted of a set of general-purpose transformations and a set of project-specific transformations. The first set comprised the transformations for the analysis of a variable and the declaration of a new variable, as well as some basic functionalities such as conversion and comparison functions. These transformations covered 278 lines of SDF code (161 productions) and 653 lines of ASF code (148 transformation rules). The implementation, testing and refinement of these transformations took 8 person days. The second set of transformations, which were specific for the Btrieve project, implemented the actual modifications and covered 118 lines of SDF code (53 production rules) and 631 lines of ASF code (74 transformation rules). The implementation, testing and refinement of these transformations took 5 person days. Hence, the effort on the transformations can be divided into time spent on basic transformations and time spent on project specific transformations.

The cursor analysis was implemented by two scripts. The first script was for extracting information from each program and consisted of 188 lines of Perl code. The second script combined the information from the first script to detect conflicting cursors, and covered 227 lines of Perl code. These scripts also had to be refined during the project in cooperation with the customer, since the exact database operations that changed the cursor were not clear to us at the beginning of the project. Also, the analysis initially yielded too many false positives and was therefore adjusted. In total, 5 person days were spent on these tools.

Then time was spent on application of the tools to the portfolio. As we mentioned

earlier, due to changing requirements most of the tools were applied several times. The figures in Table 4.5 show the time it took to apply the tools once, and the earlier attempts are covered in the development time. The parsing of the programs took 3 hours, the transformation of the programs took 5 hours, and the cursor analysis of all programs took 2 hours.

In addition to these figures, we spent time on communicating with the systems experts and management about the problem statement, technical issues, deliveries, refinements and so on. This costed about 3 person days.

To summarise, the project was carried out by two persons within four weeks. Because some of the effort was spent on generic components, such as the infrastructure, future modifications can be carried out even faster.

4.5 Conclusions

We successfully carried out the Btrieve project: a mass maintenance project, involving about 4.5 million lines of code in 45 intertwined systems. We have learned several things about automated maintenance during the project. One of the issues that was raised was typical for an IT project: during the project the requirements changed. Even though the initial problem statement appeared to be precise and clear at the start, small changes arose that influenced the project considerably. For this well-known issue, our automatic approach proved to have considerable advantages over a manual approach. In particular, it provides a consistent, flexible and low risk approach for large-scale modifications.

We deployed a mass update factory to carry out the requested modifications in a reliable and consistent way. In order to deal with the changes in the requirements, we altered our tools quickly and applied them automatically. Instead of reverting changes in tens of intertwined systems, which can be challenging when they were carried out by hand, we were able to modify our tools quickly and created a new release with transformed code. Two persons were able to carry out the project within four weeks. Another advantage is that the regular maintenance can continue, whereas with a manual approach the system needs to be frozen until the massive changes have been completed. With the automated approach, we iteratively took snapshots of the system to develop and test our tools, and refined the problem statement in consultation with the customer. When the customer and we were confident about the results, we applied the tools once more for a final release, which was tested, accepted and taken into production.

Acknowledgements This project was carried out at the Software Improvement Group Amsterdam [184] in cooperation with Steven Klusener. We are grateful to Chris Verhoef and Steven Klusener for their comments and suggestions for improvements. We would like to thank Harry Sneed and the anonymous reviewers of the journal "Science of Computer Programming" for their detailed comments. Philip Pirovano helped us with the partitioning of the call graph.

Road map We discussed how portfolio-wide changes can be carried out consistently and at low risk by using automatic tools. A consistent, automated approach reduces

chance of errors and limits the deterioration of evolving software. Nevertheless, even if small individual changes are applied in a consistent way using automatic tools, the size and complexity of software will increase. In the next chapter, we investigate what to do when an application has evolved into a complex, monolithic entity. We continue with massive modifications, but we will perform more radical changes to the code. We discuss the restructuring of Cobol programs to improve the modifiability and to support the further evolution of legacy applications.

Chapter 5

Revitalizing modifiability of legacy assets

Evolved code can be difficult to understand and modify, in particular for a programmer who is unfamiliar with the code. When the code must be changed, problems can arise. Proper evolution requires code to be modularised into loosly coupled components. Restructuring techniques can bring relief and are therefore often a prerequisite to various forms of evolution, such as wrapping, integration and replacement.

In this chapter, we discuss the modifiability of Cobol legacy code and propose an approach to improving the modifiability using automatic restructuring transformations: the Restructuring project. We present an algorithm that enables the application of these transformations to large industrial Cobol systems. The transformations were adapted to a system of 80 thousand lines of code by extending them with new transformations. We analyse and discuss the resulting source code. Two case studies (over 5 million lines of code in total) with real-life Cobol programs show the large-scale application of the transformations. This chapter is based on: N. Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance and Evolution: Research and Practice, Special issue on the European Conference on Software Maintenance and Reengineering 2003*, 16(4–5):219–254, 2004 [193, 194].

5.1 Introduction

Many organisations struggle with their legacy software systems. These systems were initiated years or even decades ago, and they are continually growing and evolving. Several programmers have been modifying the source code to keep up with changing requirements and technical evolution of the hardware platforms. Frequent changes during the system's life-cycle, carried out within tight time schedules, have resulted in complex source code, making further modifications difficult and expensive. Hence, it is not suprising that maintenance can consume up to ten times the initial development costs [26, 106, 147]. As long as the same programmers are around, these difficulties are often masked [212]. The

programmers have grown into their system over the years, making them important factors in the maintenance effort. However, they still spend a considerable amount of time on analysing the complex code before making changes. When these programmers leave the company, they take a substantial amount of knowledge with them. New programmers do not have this knowledge and up-to-date documentation is usually lacking.

At the same time, the systems are very valuable assets containing crucial business logic that has been accumulated over the years. The precise functionality of the systems is unknown and undocumented; hence, they cannot simply be replaced by new systems. Therefore, due to the complexity of these applications, some form of restructuring or partitioning is often a prerequisite for extraction, wrapping, integration, redevelopment, or replacement of such evolving software assets [180, 181, 183]. Hence, research should be continued on existing legacy, such as Cobol, which has been around for more than 40 years now. Most of the world's business runs on Cobol, and there are over 180 billion lines of Cobol code in active use, with an estimated 5 billion new lines of code added annually [8]. These figures indicate that Cobol is an important language, and existing Cobol code requires continuous modification and evolution.

In this chapter, we propose an approach for modifying Cobol assets. We briefly explained the approach in Chapter 3, where we showed that the approach can be used to combat complex and error-prone code. Here, we explain how we use and enhance restructuring transformations to revitalize the modifiability of legacy assets to allow proper evolution. The transformations are aimed to change-enable evolved Cobol code. The complex, intertwined logic of the individual programs is restructured by identifying and untangling code components. Once these components have been untangled and extracted, modifications can be made by adding new code or modifying the existing code components, without the concern of disturbing complicated programming logic. Furthermore, a system can then be ready for integration or migration; for instance, for moving to Object Cobol as described in [65], or to integrate legacy software into a service oriented architecture [181].

The Restructuring project: context and main contributions In 1998, Sellink, Sneed and Verhoef [171] worked on the restructuring of Cobol. In an industrial project, they developed a set of automatic restructuring transformations which were suitable for restructuring a specific Cobol system. They implemented the transformations using the ASF+SDF Meta-Environment [112]. Zaadnoordijk [215] migrated the transformations to a newer version of the ASF+SDF Meta-Environment [45] which was released in 2000, and he implemented a transformation algorithm which enabled the automatic application of the transformation rules.

In our project, the Restructuring project, we continued with the transformation rules and algorithm. Although this chapter continues previous work, it is self-contained. We summarise our main contributions here. We reimplemented existing transformations and improved them. Based on previous work, we improved the existing algorithm. Then we adapted the transformations to a large Cobol system by adding several extensions and new transformation rules, without interfering with the existing transformations. We analysed the transformed source code and we applied the transformations to several large industrial Cobol systems, showing the wide applicability of the ideas underlying the algorithm

presented by Sellink et al [171].

Related work Other work, which seems strongly related at first sight, is about control-flow normalisation. Restructuring in the sense of control-flow normalisation dates back to the '60s when Böhm and Jacopini [27] showed that every control-flow diagram can be transformed into an equivalent control-flow diagram, which uses only three control structures: assignment, conditional and iteration; thus without goto statements. However, Dijkstra [73] states that after removing *all* goto statements from a program, the resulting control-flow diagram cannot be expected to be more transparent than the original. We agree with his viewpoint. Work on automatic restructuring and procedure extraction [93, 120] is related to the transformation of Cobol paragraphs into simulated subroutines in this chapter. In Cobol, care should be taken when extracting paragraphs, due to the interplay of (overlapping) perform statements, fallthrough and goto statements. We also discussed this in Chapter 3.

There are a number of (commercial) restructuring tools for Cobol (e.g. [29, 60, 154], see an extensive list at [185]) but we could not find any case studies about transforming large (100,000+ lines of code) software systems using these tools. IBM has its own restructuring tool, Cobol Structuring Facility, but it has been out of service since September 2002. This tool removes all goto statements by introducing synthetic flag variables, generating nondescriptive label names, and removing existing labels. Although the tool is able to untangle very complex code, in such cases the resulting programs could not always be considered to be more modifiable. In particular, for the original maintainers, the approach can result in non-intuitive code [54]. Our approach does not introduce new variables to eliminate all goto statements, and we try to preserve the original appearance of the code as much as possible. Furthermore, our main objective is to componentise code to ease evolution; the elimination of goto statements is part of the approach.

In [95] a restructuring system is described and used in an experiment. They use flag variables and new labels to remove goto statements but they also use similar restructuring techniques as we do, such as moving paragraphs to a place without fallthrough of normal control-flow. Because the article was published in 1983, no explicit scope terminators were used since these were added to the Cobol standard in 1985. We do add explicit scope terminators because they improve the program structure and simplify other transformations. In the reported experiment several measurements were done, but the total size was not mentioned. Their largest program had 1,025 statements. We applied our transformations to large industrial systems of up to 2.6 million lines of code, our largest program had 4,000 statements.

RainCode [164] has an online restructuring service for Cobol. Their approach resembles our approach, they also use several small transformations to improve the modifiability of programs but they concentrate on replacing unstructured language constructs by structured ones, not on reducing the fallthrough of Cobol paragraphs as we do. They report on a case study which consisted of 600 generated programs. After transformation, the size of the programs was reduced to 60-90% of their original size in terms of number of statements and 49% of the programs were free of goto statements. Although they used a similar approach, these results are not comparable to ours because generated code can be excessively verbose and have many similar (unstructured) constructs. These constructs

can be replaced by structured alternatives, thereby reducing the code size significantly. We sent a small sample program to their service, and the results showed that they were able to replace many of the goto statements by structured alternatives. However, we would like to emphasise that their approach focuses on replacing unstructured language constructs by structured ones, whereas our approach focuses on untangling and extracting code to ease the modifiability and evolution of applications.

Organisation of the chapter We discuss how we revitalize Cobol assets in Section 5.2. In Section 5.3, we first present the automatic restructuring transformations and the restructuring algorithm, and then we discuss our main extensions and the adaptability of the transformations. We examine the resulting source code and also discuss some implementation issues. In Section 5.4, we evaluate the results of a large-scale application of the transformations to industrial Cobol systems. Finally, our conclusions are in Section 5.5.

5.2 Modifying Cobol programs

Cobol is the leading data processing language in the business world. The programming language has existed for over 40 years now. Since the beginning, it has undergone considerable updates and improvements. Several ANSI standards have been defined throughout the years, and many variations (dialects) to these standards have been implemented. Nowadays, Cobol also supports object-orientation. These developments are covered in [213].

In this section we discuss modifying Cobol programs for maintenance purposes. Business-critical Cobol programs have usually been modified several times by multiple programmers. These modifications, because they were carried out using different programming styles and under strict time pressure, have resulted in complex code and have led to ongoing debates on the infamous `GO TO` statement for almost 40 years now [73, 119, 168]. We will illustrate by an example how this works. The ensuing complex code is harder to modify than structured code. Then we will argue that automated restructuring using a special algorithm reduces the modifiability problems of the code, thereby revitalizing the modifiability of Cobol legacy assets.

5.2.1 The structure of a Cobol program

Despite the number of changes to the ANSI standards, the structure of Cobol programs has largely remained the same throughout the years. A program consists of four parts, called divisions, each with its own function.

- Identification division: the name of the program and programmer, modification dates, etc.
- Environment division: the description of the computer and the input/output.
- Data division: the declaration of variables and constants.
- Procedure division: the statements for coordination and computation.

```
PROCEDURE DIVISION.
01-FIRST SECTION.
PAR1.
    DISPLAY 'THIS IS THE FIRST STATEMENT'
    PERFORM PAR2
    GOBACK.
PAR2.
    DISPLAY 'THIS IS A SENTENCE'.
    ADD 1 TO A
    IF A < B
      GO TO PAR2.
```

Figure 5.1: A Cobol procedure division.

We discussed the control-flow constructs and logic of Cobol in Chapter 3 in detail. Here, we reconsider the most important notions and mechanisms that are relevant for this chapter.

The procedure division is the part were the actual programming logic takes place. It consists of *sections*. A section is denoted by a label followed by the keyword `SECTION`, and consists of *paragraphs*. A paragraph has a label and consists of *sentences*. A sentence consists of *statements* followed by a dot (`'.'`). A dot terminates the scope of all previous structured statements that have not yet been terminated (e.g. an `IF` statement). A section or paragraph label can be referred to from within the program by `PERFORM` and `GO TO` statements. A `PERFORM` statement is similar to a procedure call, a `GO TO` statement is an unconditional jump. In Figure 5.1, there is an illustrative example of a procedure division.

Control-flow of a Cobol program starts at the first statement in the procedure division and executes statements until the last statement is reached, unless a `GOBACK`, `STOP RUN` or `EXIT PROGRAM` statement is encountered. These statements end the execution of the current program. A Cobol paragraph can be executed in three ways:

- By a `GO TO` statement: control-flow is transferred to the paragraph and will go on from there to the subsequent paragraphs;
- By a `PERFORM` statement: the paragraph is executed, and the control-flow is passed back to the statement following the `PERFORM` statement;
- As a result of a *fallthrough* from a previous paragraph, this is actually an implicit `GO TO` statement. If control-flow reaches the end of a paragraph, and the paragraph was not reached by a `PERFORM` statement, control-flow will fallthrough to the subsequent paragraph (unless the last statement is a `GO TO` statement to a different label). A severe problem here is that this cannot be seen from the paragraph itself.

Three ways of executing a paragraph make it hard to understand the control-flow in Cobol programs, and even harder to modify programs. Especially the fallthrough mechanism and `GO TO` statements can be a problem for the programmer, as we shall see in greater detail later.

```
01-FILE SECTION.
OPEN-FILE.
    ...
    <fallthrough>
READ-FILE.
    ...
    <fallthrough>
CLOSE-FILE.
    ...
    <fallthrough>
FILE-EXIT.
    EXIT.
```

Figure 5.2: Initial code.

5.2.2 Modifying a program

A newly written Cobol program can be structured and designed to be modifiable by using `PERFORM` statements instead of `GO TO` statements and fallthrough. However, when the program must be modified, `PERFORM` statements are not powerful enough and a programmer must use `GO TO` statements. We will show this in an example.

Consider the code fragment in Figure 5.2. In this fragment, there is a `01-FILE SECTION` which opens, reads and closes a file. At the end, there is an exit paragraph to indicate the end of the section. All paragraphs have fallthrough to the next paragraph. Now imagine that some new error-handling code must be added to the `READ-FILE` paragraph. This code may be needed at several places so an error-handling paragraph should be implemented. The programmer who makes the change can place the paragraph somewhere in the `01-FILE SECTION` but has to take the fallthrough into account. After the error, the file must be closed by `CLOSE-FILE`. To separate the main functionality from the error handling, the programmer decides to place the paragraph at the end of the section. See Figure 5.3 for the result of the modification. Although this is a common pattern, we can reduce the number of `GO TO` statements by using a `PERFORM` statement. See Figure 5.4.

Something in the I/O changes, and we need additional error-handling code. This code must be added to the `OPEN-FILE` and `CLOSE-FILE` paragraphs. After this error code, control-flow must be passed to `FILE-EXIT`. To make things more complicated, our programmer has just retired. A new programmer will implement the modification, who decides to add the new paragraph at the end of the section. The programmer sees the `FILE-ERROR` paragraph but does not know if there is a fallthrough from `FILE-ERROR` to `FILE-EXIT`. After spending a while with the code (imagine a 1,000+ lines section), it is clear that `FILE-ERROR` is only performed by `READ-FILE` and the new paragraph can safely be added, see Figure 5.5.

Our code fragment has now been modified by two different programmers. Although their individual implementation choices appear rather structured for a Cobol program, other programmers need more and more time to understand the code. If they add new

```
01-FILE SECTION.
OPEN-FILE.
    ...
READ-FILE.
    ...
    IF BUF-SIZE > BUF-MAX
      GO TO FILE-ERROR.
    ...
CLOSE-FILE.
    ...
    GO TO FILE-EXIT.
FILE-ERROR.
    ...
    GO TO CLOSE-FILE.
FILE-EXIT.
    EXIT.
```

Figure 5.3: Code after one modification.

```
01-FILE SECTION.
OPEN-FILE.
    ...
READ-FILE.
    ...
    IF BUF-SIZE > BUF-MAX
      PERFORM FILE-ERROR
      GO TO CLOSE-FILE.
    ...
CLOSE-FILE.
    ...
    GO TO FILE-EXIT.
FILE-ERROR.
    ...
FILE-EXIT.
    EXIT.
```

Figure 5.4: Code after one modification using `PERFORM` instead of `GO TO`.

```
01-FILE SECTION.
OPEN-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
      GO TO FILE-EXIT.
    ...
READ-FILE.
    ...
    IF BUF-SIZE > BUF-MAX
      PERFORM FILE-ERROR
      GO TO CLOSE-FILE.
    ...
CLOSE-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
      GO TO FILE-EXIT.
    ...
    GO TO FILE-EXIT.
FILE-ERROR.
    ...
IO-ERROR.
    ...
FILE-EXIT.
    EXIT.
```

Figure 5.5: After two modifications.

paragraphs, they will have to figure out if there is fallthrough or not. Some programmers adhere to a style where all paragraphs have fallthrough unless there is a `GO TO` statement at the end, but this means that they cannot use `PERFORM` statements and use more `GO TO` statements instead. This will also cause a lot of time to understand the control-flow. Further modifications will cost more and more time, until it has become very difficult and expensive to modify the code.

Our example illustrated the need for `GO TO` statements when modifying a Cobol program. Relative small modifications induce adding several `GO TO` statements, and these can confuse a programmer. Moreover, it increases comprehension time. One of the first steps in program comprehension is to understand the flow of control in a program [161]; hence, new programmers, which are unfamiliar with the code, need more time before they can modify a program. Future modifications are increasingly prone to errors. Our example also illustrates the problem of modifying a modified Cobol program. The more a program has been modified, the harder it will be to modify it any further and the more `GO TO` statements are needed to express the intended logic.

5.2.3 Improving modifiability using automatic transformations

We showed the problems that arise when modifying Cobol programs. We will now present a different modification approach.

What is needed in Cobol is a subroutines section where code has no fallthrough. In this part of the program, a programmer can simply add paragraphs which are performed from a coordination section. Such an approach does not interfere with the complex control-flow. This is similar to calling functions from the '`main`' function in C, or procedure and function calls in Pascal. In these languages, separate functions are not connected by fallthrough like Cobol paragraphs. The latest Cobol standard does support user-defined functions but these functions still have to be placed in separate files, which requires moving and declaring variables if applied to legacy Cobol code. Another option is to create sub-programs, but this also requires moving and declaring variables, and to restructure sections into sub-programs, existing fallthrough logic must have been dealt with [186].

We propose an easier way for using subroutines. At the end of the program, we add a special section with only one statement: `GOBACK`. If control-flow reaches this statement, the program will end. Now we can add sections and paragraphs after this section which will never be executed by implicit control-flow (fallthrough). We will use our example code fragment to demonstrate the creation of subroutines. The special section, `BAR SECTION`, is placed at the end of the program. All code that is placed after this section will not be executed by implicit control-flow. For every section in the program, a subroutines section is created after the `BAR SECTION`. See Figure 5.6. Although the resulting program does not appear to be very different from the previous result, it is easier to modify because we did not add paragraphs that disturb the original control-flow. And a modification does not make future modifications more difficult, as the traditional approach does. We still have to introduce some `GO TO` statements, but less and they all jump to the same (exit) paragraph. The program is more in line with common ideas on structured programming, and we can modify it without having to worry about fallthrough, see Figure 5.7.

```
01-FILE SECTION.
OPEN-FILE.
    ...
READ-FILE.
    ...
CLOSE-FILE.
    ...
FILE-EXIT.
    EXIT.
...
BAR SECTION.
BAR-PARAGRAPH.
    GOBACK.
01-FILE-SUBROUTINES SECTION.
```

Figure 5.6: Adding a subroutines section.

This approach looks feasible but we may want to apply it to complex legacy Cobol programs and turn existing paragraphs also into subroutines. We can do this using the transformations developed by Sellink, Sneed and Verhoef in [171]. They developed automatic transformations to restructure Cobol programs using `GO TO` elimination and extraction of paragraphs. The idea is to restructure the code and transforming paragraphs into subroutines. This is achieved by applying several small transformations in a certain order. We discuss these transformations in Section 5.3. See Figure 5.8 for the result of applying the transformations to our code sample. The fallthrough and `GO TO` statements have been removed automatically, thereby improving the modifiability. The paragraphs in the program have been turned into loosely coupled blocks, which be moved and further changed more easily without interfering with the existing fallthrough logic. Hence, valuable Cobol assets can be modified while they remain structured, understandable and further modifiable, i.e. ready for further evolution. In the next section, we explain how to do this completely automated.

5.3 The Restructuring project: automatic code transformations

In the Restructuring project, we developed and deployed automatic transformations for the restructuring of Cobol code. We present our transformation algorithm and transformation rules. Most of the rules were taken from Sellink et al. [171] and then improved and extended. We discuss the adaptability and flexibility of the original rules by extending them with new transformation patterns and rules, which were based on a large Cobol system, and we analyse the resulting source code. We also discuss some implementation details. In the presentation of the rules, our extensions and new transformation rules have already been included; the precise extensions will be discussed afterwards.

```
01-FILE SECTION.
OPEN-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
      GO TO FILE-EXIT.
    ...
READ-FILE.
    ...
    IF BUF-SIZE > BUF-MAX
      PERFORM FILE-ERROR
      PERFORM CLOSE-FILE
      GO TO FILE-EXIT.
    ...
CLOSE-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
      GO TO FILE-EXIT.
    ...
FILE-EXIT.
    EXIT.
...
BAR SECTION.
BAR-PARAGRAPH.
    GOBACK.
01-FILE-SUBROUTINES SECTION.
FILE-ERROR.
    ...
IO-ERROR.
    ...
```

Figure 5.7: Adding two paragraphs.

```
01-FILE SECTION.
    PERFORM OPEN-FILE.
FILE-EXIT.
    EXIT.
BAR SECTION.
BAR-PARAGRAPH.
    GOBACK.
01-FILE-SUBROUTINES SECTION.
OPEN-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
    ELSE
      ...
      PERFORM READ-FILE
      PERFORM CLOSE-FILE
    END-IF.
READ-FILE.
    ...
    IF BUF-SIZE > BUF-MAX
      PERFORM FILE-ERROR
    ELSE
      ...
    END-IF.
CLOSE-FILE.
    ...
    IF FILE < 0
      PERFORM IO-ERROR
    ELSE
      ...
    END-IF.
FILE-ERROR.
    ...
IO-ERROR.
    ...
```

Figure 5.8: Automatic restructuring of the code from Figure 5.5.

5.3.1 Transformation algorithm

Our transformation algorithm was inspired by the work of Zaadnoordijk [215]. Our algorithm is illlustrated in Figure 5.9 and consists of three phases: Preprocessing, Mainprocessing and Postprocessing. First, the Preprocessing massages a program to normalise certain language constructs. This simplifies all other rules significantly, as they can make assumptions about the code. For example, after Preprocessing, every `IF` statement has a (possibly empty) `ELSE` branch and an `END-IF` scope terminator. The `ELSE` branches and the `END-IF` scope terminators simplify the search patterns of many of the other transformations. An `ELSE` branch will be removed by the Postprocessing if it is empty after Mainprocessing. Then, the Mainprocessing concentrates on two aspects: `GO TO` statements are eliminated and paragraphs that can be extracted are moved to a created subroutines section. If no more paragraphs can be extracted or no more `GO TO` statements can be eliminated, superfluous paragraph labels are eliminated in order to allow further extraction of code and elimination of `GO TO` statements. Finally, the Postprocessing cleans up the resulting code by simplifying several language constructs and by removing some of the normalisation constructs that were introduced during Preprocessing.

We informally explain that the transformation algorithm terminates. If we assume that the individual transformations terminate, the transformation algorithm will terminate, as no infinite reductions are induced [114]. We will not discuss the termination of the individual rules in detail. The Preprocessing transformations are applied once. The Mainprocessing contains two loops, of which the first will terminate, as none of the individual transformations can continue forever (e.g. there are a limited number of statements in a program that can be moved, eliminated, distributed, or simplified). Then, in the second loop, a finite number of labels can be eliminated, thereby iteratively restarting the first loop. At some point in time this process terminate. The Postprocessing transformations are also applied only once. So, assuming that the individual transformations terminate, the transformation algorithm itself terminates. In the next section, we discuss the individual rules in detail.

5.3.2 Transformation rules

We discuss the individual transformation rules here. For each rule there is a small code example that was created for clarification purposes and are therefore mostly artificial. The code examples are presented in the following format:

Transformation-name

input code	output code

Preprocessing rules

Add-labels Some versions of Cobol allow a programmer to start the Procedure division with paragraphs or statements instead of a section, and to start sections with statements instead of paragraphs. To make the code more consistent and to reduce the number of patterns in other transformation rules, Add-labels adds a missing section label and missing

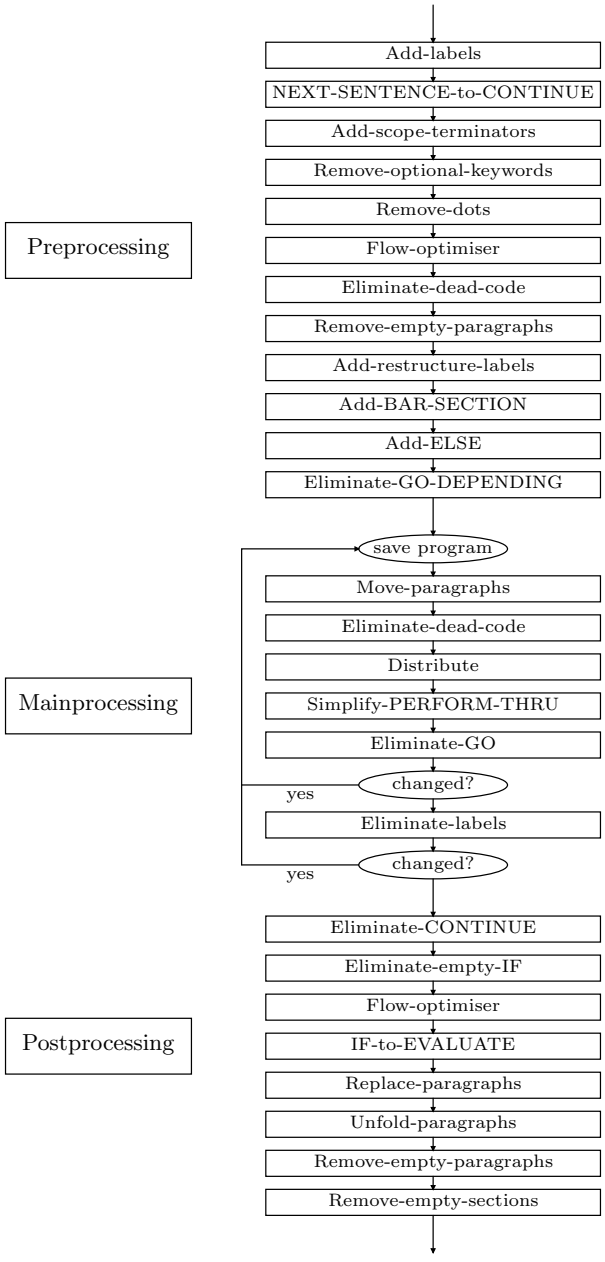

Figure 5.9: Illustrating the transformation algorithm.

paragraph labels. The first section of the program is then called 'FIRST-SECTION-OF-PROGRAM SECTION'. A missing paragraph label is called 'FIRST-PAR-OF-<section>-SECTION'. These labels can be removed after all other transformations have been applied.

Add-labels

```
PROCEDURE DIVISION.
                                  PROCEDURE DIVISION.
                                  FIRST-SECTION-OF-PROGRAM
                                  SECTION.
DISPLAYPAR.                       DISPLAYPAR.
    DISPLAY '1'.                      DISPLAY '1'.

CALCULATE SECTION.                CALCULATE SECTION.
                                  FIRST-PAR-OF-CALCULATE-SECTION.
    MOVE A TO B.                      MOVE A TO B.
```

NEXT-SENTENCE-to-CONTINUE A NEXT SENTENCE statement transfers the control-flow to the statement after the next separator dot, which is the start of the next sentence. This is actually an implicit jump which may be altered if this dot is moved or removed. We have a transformation rule, Remove-dots, which removes separator dots until each paragraph has only one dot. The dot that indicates the next sentence may be removed and therefore we transform the NEXT SENTENCE statement into a CONTINUE statement, which has no effect on the execution of a program. Any statement following the NEXT SENTENCE statement must be removed because it is dead code (code that is never executed). If we do not remove this code, it will be revived after the NEXT SENTENCE has been replaced by the CONTINUE statement.

NEXT-SENTENCE-to-CONTINUE

```
IF A > B                          IF A > B
  DISPLAY A                         DISPLAY A
ELSE                              ELSE
  NEXT SENTENCE                     CONTINUE
  DISPLAY 'This is deadcode'      .
.
```

An assumption for this transformation is that the NEXT SENTENCE statement is not used inside a structured statement (e.g. an IF statement) that has other statements between its explicit scope terminator (e.g. END-IF) and the next sentence. Otherwise the behaviour of the program may be changed by this transformation. The following example demonstrates this. Assume that both X and Y are equal to 1, the first fragment prints 'SENTENCE passed' while the second fragment prints 'Nested IF passed' and then 'SENTENCE passed'. The second DISPLAY statement causes the problem here:

NEXT-SENTENCE-to-CONTINUE (incorrect; control-flow logic is changed)

```
IF X = 1                          IF X = 1
  IF Y = 1                          IF Y = 1
    NEXT SENTENCE                     CONTINUE
  END-IF                            END-IF
  DISPLAY 'Nested IF passed'        DISPLAY 'Nested IF passed'
END-IF.                           END-IF.
DISPLAY 'SENTENCE passed'.        DISPLAY 'SENTENCE passed'.
```

Add-scope-terminators This transformation adds the explicit scope terminators. We use the Add-scope-terminators transformation rule to add an explicit ending keyword to every structured statement (i.e. statements that contain other statements) that is implicitly closed (by a dot). Other transformation rules can now assume that every structured statement is properly closed by an explicit scope terminator. This reduces the number of patterns significantly. Further, the code is more readable, as the scope of structured statements is made explicit.

Add-scope-terminators

```
IF A > B                          IF A > B
  IF A > C                          IF A > C
    DISPLAY '1'                       DISPLAY '1'
  ELSE                              ELSE
    DISPLAY '2                        DISPLAY '2'
ELSE                                END-IF
  DISPLAY '3'                     ELSE
.                                   DISPLAY '3'
                                  END-IF.
```

Remove-optional-keywords Many statements in Cobol have optional keywords; for example, an `IF` statement can have an optional `THEN` keyword, and a `GO` statement can have an optional `TO` keyword. The transformation removes these keywords in order to normalise the code and to simplify other transformations rules.

Remove-optional-keywords

```
IF A > B THEN                     IF A > B
  GO TO PAR                         GO PAR
END-IF                            END-IF
```

Remove-dots Many statements in Cobol can be terminated by a separator period at wish. Because this is not always necessary, we remove the superfluous ones using Remove-dots. A statement or a sequence of statements ended by a separator period is called a sentence. This means that one statement can form a sentence as well as ten statements. The Remove-dots transformation rule transforms the code such that each paragraph has exactly one sentence, making the code more consistent and all other transformations are simplified because the number of transformation patterns can be reduced.

Remove-dots

```
DISPLAY A.                        DISPLAY A
DISPLAY B.                        DISPLAY B.
```

Removing the superfluous dots can only be done if all `NEXT SENTENCE` statements are removed, otherwise the behaviour of the code can be different. The following code example demonstrates this. The first fragment displays '`Next sentence`', whereas the second fragment skips the display statement.

Remove-dots (incorrect; control-flow logic is changed)

```
NEXT SENTENCE.                    NEXT SENTENCE
DISPLAY 'Next sentence'.          DISPLAY 'Next sentence'.
```

Flow-optimiser This transformation optimises the control-flow of `IF` statements. Nested `IF` statements without `ELSE` branches and no statements except in the inner `IF` are merged into one `IF` statement. The conditions are `AND`ed into one condition. This reduces the number of patterns for other transformation rules and thus allows more code to be transformed. Merging these `IF` statements may result in complex conditions and therefore in such cases these `IF` statements could be transformed back into their original state after application of all transformations.

Flow-optimiser

```
IF X = 1                          IF X = 1 AND Y = 2
  IF Y = 2                          DISPLAY '12'
    DISPLAY '12'                  END-IF
  END-IF
END-IF
```

Eliminate-dead-code This transformation removes code that is never executed. This is code that is located after an unconditional jump instruction (`GO TO` statement) or a paragraph that is never executed. Paragraphs that are never executed are also removed. This rule is also used in the Mainprocessing of our algorithm because we move code around and that may reveal dead code.

Eliminate-dead-code

```
DISPLAY '1'                       DISPLAY '1'
GO PAR                            GO PAR
DISPLAY '2'
```

Remove-empty-paragraphs This transformation removes empty paragraphs. All `GO TO` references to the paragraph are replaced by references to the following paragraph. `PERFORM THRU` references are replaced by the appropriate label, and all other `PERFORM` references are removed. The transformation normalises the code and also simplifies other transformations. In the code example, the empty paragraph `PAR2` is removed. The reference in the `PERFORM THRU` is shifted, the `PERFORM PAR2` is completely removed, and `GO PAR2` is shifted to `GO PAR3`. The resulting `PERFORM THRU` statement can be simplified to `PERFORM PAR3` by Simplify-PERFORM-THRU during Mainprocessing.

Remove-empty-paragraphs

```
PAR1.                             PAR1.
    PERFORM PAR2 THRU PAR3            PERFORM PAR3 THRU PAR3
    PERFORM PAR2                      GO PAR3.
    GO PAR2.
PAR2.
PAR3.                             PAR3.
    DISPLAY '1'.                      DISPLAY '1'.
```

Add-restructure-labels One of the aims of the restructuring is to extract paragraphs by moving `GO TO`-free paragraphs behind the `BAR SECTION`. This is done by Move-paragraphs during Mainprocessing, which replaces a paragraph by a `PERFORM` statement

referencing this paragraph. However, the transformation will not match on the first paragraph of a section because a paragraph is required were the PERFORM statement can be placed. Therefore, we add a (temporary) dummy label to allow the first paragraph of a section to be extracted. This label can be removed after all other transformations have been applied.

Add-restructure-labels

```
EXAMPLE SECTION.                    EXAMPLE SECTION.
                                    RESTRUCTURE-PAR.
PAR1.                               PAR1.
    DISPLAY '1'.                        DISPLAY '1'.
```

Add-BAR-SECTION This transformation adds a new section at the end of a program, the BAR SECTION. This section contains one paragraph with a GOBACK command. A programmer can then safely add new paragraphs here and use PERFORM statements to execute them. Any code that is placed after this section will never be executed by implicit control-flow because it is only referenced by PERFORM statements. For each section of the program, we add a subroutines section behind this BAR SECTION. After that, we can move paragraphs to these subroutines sections that are then performed. This is done by Move-paragraphs in the Mainprocessing.

Add-BAR-SECTION

```
EXAMPLE SECTION.                    EXAMPLE SECTION.
PAR.                                PAR.
    DISPLAY '1'.                        DISPLAY '1'.

                                    BAR SECTION.
                                    BAR-PARAGRAPH.
                                        GOBACK.

                                    EXAMPLE-SUBROUTINES SECTION.
```

Add-ELSE This transformation adds an empty ELSE branch to every IF statement without an ELSE branch. This rule reduces the number of patterns in the other transformation rules. Each transformation rule can now assume that every IF statement has a (possibly empty) ELSE branch. ELSE branches that are still empty after the Mainprocessing will be removed by the Eliminate-empty-IF transformation rule in the Postprocessing.

Add-ELSE

```
IF A > B                            IF A > B
  DISPLAY '1'                         DISPLAY '1'
END-IF                              ELSE
                                    END-IF
```

Eliminate-GO-DEPENDING This transformation transforms a GO DEPENDING statement into an EVALUATE statement. Although we introduce new GO TO statements, these statements can be transformed by other transformation rules whereas the GO DEPENDING cannot.

Eliminate-GO-DEPENDING

```
GO PAR1                          EVALUATE X
   PAR2 DEPENDING X                WHEN 1 GO PAR1
                                   WHEN 2 GO PAR2
                                 END-EVALUATE
```

Mainprocessing rules

The transformation rules of the Mainprocessing are applied in two different loops until no more code can be transformed. The first loop focuses on code extraction by Move-paragraphs and on `GO TO` elimination by Eliminate-GO. The second loop applies Eliminate-labels and is done if no other Mainprocessing transformation can be applied. This transformation merges consecutive paragraphs in order to trigger Move-paragraphs and Eliminate-GO again.

Move-paragraphs This transformation extracts paragraphs and transforms them into subroutines. Paragraphs that are free of external `GO TO` statements (the single-entry-single-exit property) are moved behind the `BAR SECTION`. In the example, the paragraph `PAR2` is extracted and replaced by a `PERFORM` statement, and all `GO TO` statements to this paragraph are replaced by a `PERFORM` followed by a `GO TO` statement to paragraph `PAR3`. This is called *`GO TO`-shifting*. Note that, to preserve the original logic, `PAR1` may not be referenced by a `PERFORM` statement, otherwise the behaviour of the program can be different.

Move-paragraphs

```
EXAMPLE SECTION.                 EXAMPLE SECTION.
PAR1.                            PAR1.
    DISPLAY '1'.                     DISPLAY '1'
PAR2.                                PERFORM PAR2.
    DISPLAY '2'.
PAR3.                            PAR3.
    GO PAR2.                         PERFORM PAR2
                                     GO PAR3.
BAR SECTION.                     BAR SECTION.
BAR-PARAGRAPH.                   BAR-PARAGRAPH.
    GOBACK.                          GOBACK.

EXAMPLE-SUBROUTINES SECTION.     EXAMPLE-SUBROUTINES SECTION.
                                 PAR2.
                                     DISPLAY '2'.
```

Distribute This transformation optimises `IF` statements. Statements that occur in both branches of an `IF` statement are transformed such that one occurrence is sufficient. This rule can be very effective if, for instance, a `GO TO` statement is placed outside the `IF` statement. Such a `GO TO` statement may then be eliminated with the Eliminate-GO transformation rule. These `IF` statements with similar `GO TO` statements in both branches can be the result of `GO TO`-shifting (see Move-paragraphs).

Distribute

```
IF A > B                          IF A > B
  DISPLAY '1'                       DISPLAY '1'
  DISPLAY '3'                     ELSE
ELSE                                DISPLAY '2'
  DISPLAY '2'                     END-IF
  DISPLAY '3'                     DISPLAY '3'
END-IF
```

Simplify-PERFORM-THRU The complex logic of a `PERFORM THRU` statement can prevent matching of transformations. For instance, Move-paragraphs cannot extract the first or last paragraph of a `PERFORM THRU` block. Therefore, we have the transformation Simplify-PERFORM-THRU to simplify these statements as much as possible. If the first or the last paragraph in such a block is not used as a reference inside the block, it can be executed by a normal `PERFORM` statement instead. In the code example below, `PERFORM PAR1 THRU PAR3` can be simplified to `PERFORM PAR1 PERFORM PAR2 THRU PAR3` since `PAR1` is not used as a reference inside the block. Paragraphs `PAR2` and `PAR3`, on the other hand, cannot be simplified by this transformation because there is a `GO TO` statement in `PAR3` that references `PAR2`. Now Move-paragraphs can extract `PAR1` because it is separated from the complex `PERFORM THRU` block.

Simplify-PERFORM-THRU

```
PAR0.                             PAR0.
    PERFORM PAR1 THRU PAR3.           PERFORM PAR1
                                      PERFORM PAR2 THRU PAR3.
PAR1.                             PAR1.
    DISPLAY '1'.                      DISPLAY '1'.
PAR2.                             PAR2.
    DISPLAY '2'.                      DISPLAY '2'.
PAR3.                             PAR3.
    DISPLAY '3'                       DISPLAY '3'
    GO PAR2.                          GO PAR2.
```

Eliminate-GO The Eliminate-GO transformation can remove several types of `GO TO` statements. In some cases, a small amount of code is duplicated to remove a `GO TO` statement. If more than three statements must be duplicated to remove a `GO TO` statement, or one of the statements to be duplicated is a `GO TO` statement, the transformation does not match (to avoid complex structures and to enforce termination). This heuristics can easily be adjusted if necessary. In the example, `DISPLAY '4'` is duplicated to transform a `GO TO` statement into an in-line `PERFORM` statement (similar to a while-loop). The performed labels in the program are taken into account in order to make sure that the existing logic remains the same.

Eliminate-GO

```
PAR1.                              PAR1.
    IF A > B                           IF A > B
      DISPLAY '1'                        DISPLAY '1'
      GO PAR2                          ELSE
    ELSE                                 DISPLAY '2'
      DISPLAY '2'                        DISPLAY '3'
    END-IF                             END-IF.
    DISPLAY '3'.
PAR2.                              PAR2.
    DISPLAY '4'                        DISPLAY '4'
    IF A > B                           PERFORM TEST UNTIL NOT (A>B)
    DISPLAY '5'                          DISPLAY '5'
      GO PAR2                            DISPLAY '4'
    ELSE                               END-PERFORM
      DISPLAY '6'                      DISPLAY '6'.
    END-IF.
```

Eliminate-labels This transformation removes unused paragraph labels. The idea is to merge paragraphs that are always executed sequentially. This may indicate that their functionality is related. If a paragraph is not referenced by a `GO TO` statement or `PERFORM` statement, its body is concatenated to its predecessor. We must also check that the preceding paragraph is not referenced by a `PERFORM` statement because merging such paragraphs changes the behaviour of the program. This transformation is done if no Mainprocessing transformation can be applied.

Eliminate-labels

```
PAR1.                              PAR1.
    DISPLAY '1'.                       DISPLAY '1'
PAR2.                                  DISPLAY '2'.
    DISPLAY '2'.
```

Postprocessing rules

Eliminate-CONTINUE This transformation removes the superfluous `CONTINUE` statements. In general, a `CONTINUE` statement has no effect on the execution of a program and can be removed. Branches of an `IF` statement can be empty after the Mainprocessing transformations. Therefore, we have Eliminate-empty-IF to simplify or remove such `IF` statements. If a `CONTINUE` statement is the only statement in an `EVALUATE` branch, it must remain otherwise control-flow will fallthrough to the next branch.

Eliminate-CONTINUE

```
IF A > B                          IF A > B
  CONTINUE                        END-IF
END-IF

EVALUATE X                        EVALUATE X
  WHEN 1 CONTINUE                   WHEN 1 CONTINUE
  WHEN 2 DISPLAY '2'                WHEN 2 DISPLAY '2'
         CONTINUE
END-EVALUATE                      END-EVALUATE
```

Eliminate-empty-IF This transformation removes empty `IF` statements and empty `IF` and `ELSE` branches. After elimination of `GO TO` statements or removal of the `CONTINUE` statements by Eliminate-CONTINUE, an `IF` statements or its `IF` and `ELSE` branch can be empty. Such statements and empty branches are eliminated.

Eliminate-empty-IF

```
IF A > B                          IF NOT (A > B)
ELSE                                DISPLAY '1'
  DISPLAY '1'                     END-IF
END-IF

IF A > C                          IF A > C
  DISPLAY '2'                       DISPLAY '2'
ELSE                              END-IF
END-IF

IF A > D
ELSE
END-IF
```

IF-to-EVALUATE This transformation transforms (deeply) nested `IF-ELSE-IF` statements to the more structured `EVALUATE` statement. The nested `IF` statements were either programmed on purpose to simulate an `EVALUATE` statement or they were the result of `GO TO` elimination.

IF-to-EVALUATE

```
IF A > B                          EVALUATE TRUE
  DISPLAY '1'                       WHEN A > B
ELSE                                  DISPLAY '1'
  IF B > C                          WHEN B > C
    DISPLAY '2'                       DISPLAY '2'
  ELSE                              WHEN C > D
    IF C > D                          DISPLAY '3'
      DISPLAY '3'                   WHEN OTHER
    ELSE                              DISPLAY '4'
      DISPLAY '4'                 END-EVALUATE
    END-IF
  END-IF
END-IF
```

Replace-paragraphs This transformation searches for paragraphs behind the `BAR SECTION` that have the same body. All but one are removed and all references to the removed paragraphs are then replaced by references to the remaining paragraph.

Replace-paragraphs

```
EXAMPLE SECTION.                    EXAMPLE SECTION.
PAR1.                               PAR1.
    PERFORM PAR2                        PERFORM PAR2
    PERFORM PAR3.                       PERFORM PAR2.

BAR SECTION.                        BAR SECTION.
BAR-PARAGRAPH.                      BAR-PARAGRAPH.
    GOBACK.                             GOBACK.

EXAMPLE-SUBROUTINES SECTION.        EXAMPLE-SUBROUTINES SECTION.
PAR2.                               PAR2.
    DISPLAY 'X'.                        DISPLAY 'X'.
PAR3.
    DISPLAY 'X'.
```

Unfold-paragraphs This transformation replaces indirect paragraph calls to direct paragraph calls. This means that a paragraph behind the `BAR SECTION` which only performs another paragraph is removed, and every reference is replaced by a reference to the other paragraph. This transformation also shifts `PERFORM` statements. A `PERFORM` statement at the end of a paragraph is moved to the place where this paragraph is performed. This is done in order to move these coordination statements before the `BAR SECTION`.

Unfold-paragraphs

```
EXAMPLE SECTION.                    EXAMPLE SECTION.
PAR1.                               PAR1.
    PERFORM PAR2.                       PERFORM PAR3
                                        PERFORM PAR4.
BAR SECTION.                        BAR SECTION.
BAR-PARAGRAPH.                      BAR-PARAGRAPH.
    GOBACK.                             GOBACK.

EXAMPLE-SUBROUTINES SECTION.        EXAMPLE-SUBROUTINES SECTION.
PAR2.
    PERFORM PAR3.
PAR3.                               PAR3.
    DISPLAY '3'                         DISPLAY '3'.
    PERFORM PAR4.
PAR4.                               PAR4.
    DISPLAY '4'.                        DISPLAY '4'.
```

Remove-empty-sections In the Preprocessing, for each section a subroutines section is created where paragraphs can be moved. In some cases, these subroutines sections remain empty because no paragraphs were moved. This transformation removes such sections.

Remove-empty-sections

```
BAR SECTION.                          BAR SECTION.
BAR-PARAGRAPH.                        BAR-PARAGRAPH.
    GOBACK.                               GOBACK.

EXAMPLE-SUBROUTINES SECTION.
```

5.3.3 Adaptability and main extensions

The original transformation rules from Sellink et al. [171] were created for transforming a specific system. All `GO TO` statements were removed from that system. In the Restructuring project, we adapted the transformations to a different, larger system of approximately 80,000 lines of code IBM Cobol by extending the transformations. This system served as our source-base and was used to calibrate the transformations in a way they maximise code extraction and `GO TO` elimination. We aimed at maximising these aspects while preserving the original appearance of the code as much as possible. This means that we cannot achieve a `GO TO` elimination percentage of 100% because this requires introduction of variables and more code duplication. Furthermore, as we pointed out in Chapter 3, the existence of Cobol minefields may hinder `GO TO` elimination. We also cannot extract 100% of the code because `PERFORM` will appear before the `BAR SECTION` to reference extracted paragraphs. We extended the transformations in four steps:

1. we generalized the existing Eliminate-GO patterns;
2. we extended the Preprocessing transformations;
3. we added three new Eliminate-GO patterns;
4. we improved extraction of `GO TO`-free code.

First, we generalized the Eliminate-GO transformation. We made several changes and improvements. Some of the existing patterns handled only `IF` statements without `ELSE` branches. We added the Add-ELSE transformation to provide every `IF` statement with an `ELSE` branch. Then we modified the existing patterns such that they assume each `IF` statement has an `ELSE` branch. This way, we unified 30 `GO TO` elimination patterns into 20 while covering the same code patterns. We also allowed some of the rules to duplicate certain statements in order to eliminate a `GO TO` statement, which was not necessary for the `GO TO` statements of the system used by Sellink et al. [171]. We used a heuristic that no more than three statements may be duplicated to remove a `GO TO`, and these three statements may not contain a `GO TO` statement (a statement containing other statements is counted as several statements).

Second, we added and improved several Preprocessing transformations. This way the code was in better shape for Mainprocessing and thus improved matching of other transformation rules. For example, we added all explicit scope terminators. In the original rules, only the `END-IF` was added. Examination of the source code revealed that many structured statements were implicitly closed by a dot (e.g. the `READ` statement), resulting in paragraphs with more than one sentence. Several of the transformation rules assume that there is only one sentence, and therefore do not match these paragraphs. After adding

```
Eliminate-GO(
  IF Condition                IF Condition
    Statements1                 Statements1
    GO Label                  ELSE
  ELSE              ⟶          Statements2
    Statements2                 Statements3
  END-IF                      END-IF
  Statements3                 GO Label
  GO Label
)
```

Figure 5.10: Added Eliminate-GO transformation pattern.

these explicit scope terminators (see the transformation Add-scope-terminators), the implicit scope terminators can be removed. Another example of the improvement is the removal of optional keywords. We added the Remove-optional-keywords transformation for this purpose. Without these keywords, many transformation patterns matched more often and were simplified. Furthermore, we removed empty paragraphs using Remove-empty-paragraphs. This way, other transformation rules can assume that paragraphs are never empty and therefore matched more often and their patterns were simplified. In addition to these improvements, there were numerous smaller improvements.

Third, we added new Eliminate-GO patterns. To find out whether it was paying off to implement a new pattern, we implemented a pattern counting tool to count how often a certain code pattern occurred in the source code. Using this tool, we identified several interesting `GO TO` elimination patterns. This led to the addition of three simple patterns, one of which is shown in Figure 5.10. The shown pattern can shift a `GO TO` statement to the end of a paragraph and this `GO TO` statement may be placed closer to the label it refers to. There is one `GO TO` statement less and the remaining `GO TO` statement may be removed by one of the other patterns.

Finally, we improved the extraction of code. We added three new transformation patterns to Move-paragraphs. For example, a paragraph can also be extracted if it has a `GO TO` statement at its end. The `GO TO` statement will be shifted to the `PERFORM` statements that reference this paragraph. Figure 5.11 shows a code example of this transformation. `PAR2` is extracted and `GO AWAY` is added after each `PERFORM PAR2`. We also added the Add-restructure-labels transformation, which allows the first paragraph of a section to be extracted. Furthermore, we allowed Move-paragraphs to extract paragraphs which had only local `GO TO` statements since these paragraphs are not connected to other paragraphs by `GO TO` statements and can be moved freely. In Section 5.3.4 we will show an example of a Cobol section that has been restructured completely by extracting all paragraphs.

To calibrate the transformations, we measured the percentage of extracted statements and eliminated `GO TO` statements after a transformation run. The source-base consisted of 87 IBM Cobol programs covering a total of approximately 80,000 lines of code, and was taken from a real-life system of a bank. The results are summarised in Table 5.1. In the first run, which was done with the original transformation rules, we achieved an

Move-paragraphs

```
EXAMPLE SECTION.                    EXAMPLE SECTION.
PAR1.                               PAR1.
    DISPLAY '1'.                        DISPLAY '1'
PAR2.                                   PERFORM PAR2
    DISPLAY '2'                         GO AWAY.
    GO AWAY.
...                                 ...
PAR5.                               PAR5.
    PERFORM PAR2.                       PERFORM PAR2
                                        GO AWAY.
...  GOBACK                         ...  GOBACK
EXAMPLE-SUBROUTINES SECTION.        EXAMPLE-SUBROUTINES SECTION.
                                    PAR2.
                                        DISPLAY '2'.
```

Figure 5.11: Move-paragraphs code example of the extension.

elimination percentage of 54% `GO TO` statements (1,598 of 2,938 statements) and we extracted 24% of the statements. This first result was used as a baseline. After we generalised the `GO TO` elimination patterns, 65% of the `GO TO` statements was removed and 33% of the statements was extracted. Extension and improvement of the Preprocessing transformations raised the elimination percentage to 70% and the extraction percentage to 39%. After we added the new `GO TO` elimination patterns, we achieved an elimination percentage of 76% and an extraction percentage of 46%. In the final run, with the improved extraction patterns, we managed to extract 79% of the statements and eliminated 82% of the `GO TO` statements.

The extensions to the original transformations were successful. We added new patterns but extensive alterations to the original transformations were not necessary. Our results show that the existing transformations can be adapted and applied to a system, and do not interfere with the existing transformations; this agrees with the findings in [83] where the notion of conservative extension of a rewrite system was used for software renovation. Conservative means that new patterns do not interfere with the existing patterns, and this can be achieved if patterns do not overlap or if appropriate conditions are added. We carefully added new rules, and in case of overlapping patterns, we added specific conditions to avoid interference with the existing rules. Summarising, the restructuring approach is quite suitable for adaptation to different systems.

5.3.4 Transformed code of the source-base

We measured the eliminated `GO TO` statements and the extracted statements to calibrate the transformations on the source-base. We will now discuss other properties as well and examine the resulting code.

Table 5.2 shows more details of the transformed source-base after adaptation of the transformations. The `GO TO` statements were classified as exit, local, backward or for-

Table 5.1: Results of the adaptation.

Source-base (80,000 LOC and 2,938 `GO TO` statements)	eliminated `GO TO`s	extracted statements
First run	54.4%	24.3%
Generalized `GO TO` elimination patterns	64.5%	32.5%
Extended Postprocessing transformations	70.0%	39.1%
New `GO TO` elimination patterns	75.5%	45.6%
New and improved extraction patterns	81.9%	78.7%

Table 5.2: Results of the source-base.

Source-base	Before	After	Changed
statements	32,742	32,591	- 0 %
`GO TO` statements	2,938	532	- 82 %
forward `GO TO` statements (not to `EXIT`)	1,612	38	- 98 %
exit `GO TO` statements	565	240	- 58 %
local `GO TO` statements	490	234	- 52 %
backward `GO TO` statements	271	20	- 93 %
in-line `PERFORM` statements	0	420	N.A.
out-of-line `PERFORM` statements	2,191	5,180	+ 136 %
`IF` statements	4,836	3,254	- 33 %
`EVALUATE` statements	0	320	N.A.
percentage of extracted statements	0 %	79 %	+ 79 %

ward. Exit means that a `GO TO` statement jumps to an `EXIT` paragraph, which is a paragraph that contains only an `EXIT` statement and is very often located at the end of a section. The labelnames of such paragraphs have often `EXIT` as a postfix. Local means that a `GO TO` statement jumps within a paragraph. Backward means that the `GO TO` statement jumps to a paragraph preceding the paragraph that contains the `GO TO` statement. Forward means that the `GO TO` statement jumps forward but not to an `EXIT` paragraph. The backward and forward `GO TO` statements can cause most complexity in code since it is often unclear where they jump to. Before transformation, the majority of the `GO TO` statements jumped forward. After the transformation, many of the remaining `GO TO` statements were local (44% of total) or exit (45% of total). The rest of them was either backward (4% of total) or forward (7% of total). The forward `GO TO` statements were removed by `GO TO`-shifting, where the `GO TO` statement was shifted each time a little closer to the paragraph it jumped to and finally it was removed. We will examine some of the remaining `GO TO` statements later.

The number of statements was reduced by one percent. Statements were removed, which can be due to the following: dead code removal, unfolded paragraphs, replacement of paragraphs, `GO TO` elimination, removal of `CONTINUE` statements, and transforma-

tion IF statements to an EVALUATE statement. Statements were added, which can be due to the following: addition of PERFORM statements for the extracted paragraphs, some code duplication to remove certain GO TO statements, shifted PERFORM statements, and transformation of GO DEPENDING statements to EVALUATE statements with GO TO statements.

The number of in-line PERFORM statements increased from 0 to over 400. This statement is the equivalent of the 'while'-loop in other programming languages. The out-of-line PERFORM statements increased by 136%; this statement was used for executing the extracted paragraphs. It is the equivalent of a method or procedure call in other programming languages. The number of IF statements decreased by one third because of replacement by in-line PERFORM statements and EVALUATE statements, and because of merged IF statements by Flow-optimiser. The number of EVALUATE statements increased because they replaced GO DEPENDING statements and nested IF statements.

We extracted 79% of the statements. In Figure 5.12, we see a Cobol section from the source-base where all paragraphs have been extracted. The references to these paragraphs are placed in RESTRUCTURE-PAR, which was added by Add-restructure-labels. This label can also be removed if desired; it is not referenced. In the example, 16 of the 21 statements were extracted, which is 76%. The exit-paragraph 7299 can also be extracted or removed but it is left in the section for documentation purposes. The resulting code shows a clear separation between the coordination and computation: perform statements indicate what should be done, and the paragraphs in the subroutines section carry out the actual computations. For control-flow graphs of the code and of an entire restructured program, we refer to Figure 3.16 and Figure 3.20 in Chapter 3.

We will now examine the remaining GO TO statements more closely. We see from the results in Table 5.2 that about 20% of the GO TO statements were not removed from the source-base, and therefore some of the code could not be extracted. As we observed earlier, most of the remaining GO TO statements are either local or exit GO TO statements. Exploring the source code revealed why these statements were not eliminated. Several of these GO TO statements represent multi-exit loops, and it requires code duplication and introduction of variables to remove them. It is not likely that removing them will increase the modifiability. The exit GO TO acts often as a break statement, as known in languages like C and Java, and the local GO TO sometimes acts as a continue statement in C or Java. We could add comments to these statements to indicate that they are a break or a loop jump. In Figure 5.13, a code fragment before and after transformation is shown. A loop is used to read records from a file, and an exit GO TO is used as a break. When INDEX is greater than 99 before the end-of-file is reached, the loop is terminated and an error message is displayed. When the end-of-file is reached, control-flow jumps out of the loop using a GO TO. Note that paragraph 1097 was extracted, and therefore not shown. This example shows a typical GO TO statement that was not eliminated from the code, as we chose not to introduce additional variables. If it was deemed necessary to eliminate such GO TO statements, this can be implemented without any problem.

Other remaining GO TO statements were often multiple local loops, several nested loops within one paragraph or nested loops over different paragraphs. These loops can be the result of many modifications, and in order to remove them it requires duplication of code (more than three statements) and introduction of flag variables. Such code should be

```
72-PT-MS SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC.
  MOVE 0 TO PT-M-TABLE-R.
7203.
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC < PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC = PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  SUBTRACT PERIOD-ACC FROM M-ACC.
  GO TO 7203.
7205.
  ADD PERIOD-ACC TO M-ACC.
  IF M-ACC > 72
    GO TO 7207.
  MOVE 1 TO M-TABLE(M-ACC).
  GO TO 7205.
7207.
  MOVE PT-M-TABLE-R TO PT-MS.
  MOVE 0 TO DEP-DATA IN TRA-NUM.
  IF P-TYPE
    GO TO 7299.
7209.
  MOVE 0 TO MIN-AMOUNT  IN TRA-NUM
            MAX-AMOUNT  IN TRA-NUM
            REMAIN-PERC IN TRA-NUM.
7299.
  EXIT.
```

```
72-PT-MS SECTION.
RESTRUCTURE-PAR.
  PERFORM 7201
  PERFORM 7203
  PERFORM 7205
  PERFORM 7207.
7299.
  EXIT.
  ... GOBACK.
72-PT-MS-SUBROUTINES SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC
  MOVE 0 TO PT-M-TABLE-R.
7203.
  PERFORM TEST BEFORE UNTIL ( M-ACC < 01 )
                OR ( M-ACC < PERIOD-ACC )
                OR ( M-ACC = PERIOD-ACC )
    SUBTRACT PERIOD-ACC FROM M-ACC
  END-PERFORM
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
  END-IF
  MOVE 1 TO M-TABLE(M-ACC).
7205.
  ADD PERIOD-ACC TO M-ACC
  PERFORM TEST BEFORE UNTIL ( M-ACC > 72 )
    MOVE 1 TO M-TABLE(M-ACC)
    ADD PERIOD-ACC TO M-ACC
  END-PERFORM.
7207.
  MOVE PT-M-TABLE-R TO PT-MS
  MOVE 0 TO DEP-DATA IN TRA-NUM
  IF NOT ( P-TYPE )
    PERFORM 7209
  END-IF.
7209.
  MOVE 0 TO MIN-AMOUNT  IN TRA-NUM
            MAX-AMOUNT  IN TRA-NUM
            REMAIN-PERC IN TRA-NUM.
```

Figure 5.12: Restructuring of a Cobol section.

```
1001.
  ADD 1 TO INDEX.
  IF INDEX > 99
    GO TO 1097.
  MOVE PRICE TO TABLE01 (INDEX).
1003.
  READ FILE01 INTO PRICE
    AT END GO TO 1099.
  GO TO 1001.
1097.
  DISPLAY 'TABLE ERROR'.
1099.
  EXIT.
```

```
1001.
  ADD 1 TO INDEX
  PERFORM TEST BEFORE UNTIL (INDEX > 99)
    MOVE PRICE TO TABLE01 (INDEX)
    READ FILE01 INTO PRICE
      AT END GO 1099
    END-READ
    ADD 1 TO INDEX
  END-PERFORM
  PERFORM 1097.
1099.
  EXIT.
```

Figure 5.13: The reading of a file, before and after the transformation.

restructured with the help of a system expert.

5.3.5 Implementation details

The transformation rules and the transformation algorithm were both implemented using the ASF+SDF Meta-Environment [112, 45], We refer to Chapter 2 for detailed information on the use of ASF+SDF and the ASF+SDF Meta-Environment for source code transformations. Here, we explain how several ASF+SDF specifications can be combined to create tools for source code transformations, such as the implementation of the restructuring algorithm. We briefly touch upon the ASF+SDF Meta-Environment and we give some details about the size of our implementation. Before the Cobol code was transformed, it was parsed. We will not discuss the preprocessing and postprocessing phases before and after parsing, as that has been covered in Chapter 2 and Chapter 4. We also say something about the performance of the transformations in terms of application time, and we give a detailed description of how we dealt with the original comments in the source code.

Using the ASF+SDF Meta-Environment to create transformation tools

An ASF+SDF specification consists of a collection of modules, and each module has an SDF part and optionally an ASF part. A specification is used to generate parse- and equations tables, which are then used for the actual transformation. An overview of this process is given in Figure 5.14; note that the pre- and postprocessing of the Cobol code has been omitted. The parse table is used by the parser to build a parse tree from an (input) text file. The equations, or rewrite rules, can be applied to the parse tree using the ASF interpreter, or they can be compiled to C code using the ASF+SDF compiler [32] and then to native code (using a C compiler). The compiled binary can directly be applied to the parse tree. The use of the ASF compiler instead of the ASF interpreter has a significant performance advantages, but it also has some drawbacks, as the current implementation does not fully support all features of ASF+SDF and preserve layout. We briefly address these issues later. We also made extensive use of the incorporated traversal capabilities [35, 40, 44], which can reduce the manual effort of writing transformations [40, 44].

Grammar and rewrite rules

Our specification consisted of about 10 modules for the Cobol grammar and about 30 modules for the transformations. The grammar we used was a subset of the IBM VS Cobol II grammar [129]. The full grammar was recovered from a reference manual [131] and then a tolerant grammar was derived [116]. The subset had approximately 780 production rules; this was about 930 lines of SDF code. We reused ASF code from Zaadnoordijk [215] and Sellink et al. [171] to implement the transformations rules. Our transformation algorithm was also fully implemented in ASF+SDF. In total, our transformations consisted of about 200 production rules, covering almost 1,200 lines of SDF code. This SDF code was for specifying the signature of the transformations and for declaration of the variables that were used in the transformation patterns. The actual transformation consisted of 230 rewrite rules, covering almost 2,800 lines of ASF code.

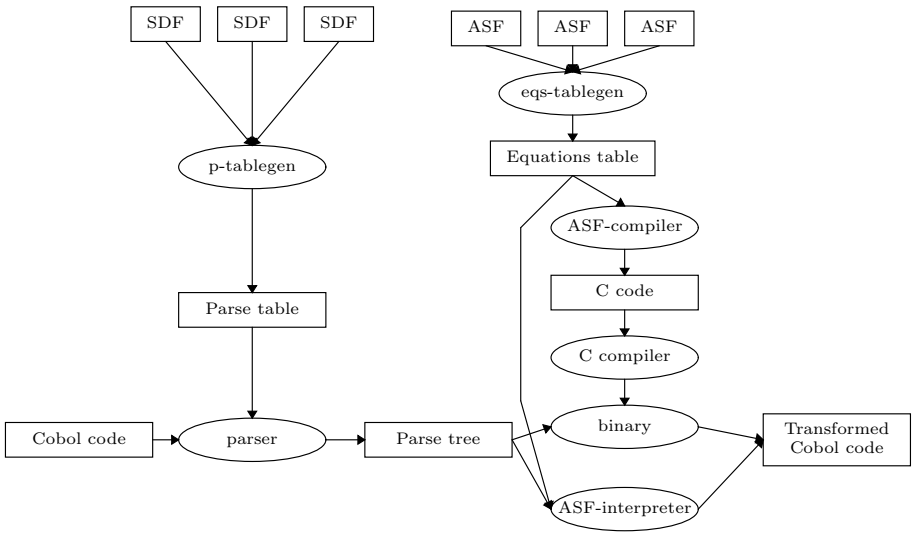

Figure 5.14: From specification to transformation using the ASF interpreter or the ASF compiler.

An example of some ASF code that was used in one of the transformations is shown in Figure 5.15. This code is part of the Move-paragraphs transformation. We briefly discuss the code. The * is used to denote zero or more occurrences, the ? represents optionals and % is used for comments in ASF+SDF. The condition for the rewrite rule is written above the ===> and the rewrite relation is denoted by the =. The traversal function `move-paragraphs()` traverses the program until the left-hand side of the equation is found. Labels that are referenced by `PERFORM` statements (`Perform-labels` and `Perform-thru-labels`) are supplied as extra arguments and are determined before traversing begins (not shown). A paragraph that contains no global `GO TO` statements (`Label-name2`) is moved to the corresponding subroutines section and a `PERFORM` statement is added to the preceding paragraph. The subroutines section is found using `is-subroutines-section()` to compare the current section label `Section-label1` with the target section label `Section-label2`. The paragraph to be moved may not be referenced by a `PERFORM THRU` statement and the preceding paragraph (`Label-name1`) may not be referenced by a `PERFORM` or a `PERFORM THRU` statement. All `GO TO` statements references to the moved paragraph are replaced by a `PERFORM` statement and a `GO TO` statement to the next paragraph (`Label-name3`); this is done by `shift-goto()`, which is also a traversal function.

Performance

We used an AMD Athlon 2200+ MHz with 512 Megabytes memory running Linux. Generation of the parse table took 20 seconds. Generation of the equations table took about 27 minutes; during the generation process, a separate parse table is generated for each

```
[rule-move-paragraphs]
   contains-no-global-goto( Label-name2. Statement*2. ) = true,
   is-subroutines-section( Section-label1, Section-label2 ) = true
   occurs-in( Label-name1, Perform-labels ) = false,
   occurs-in( Label-name1, Perform-thru-labels ) = false,
   occurs-in( Label-name2, Perform-thru-labels ) = false,
   ===>
   move-paragraphs(
     Section*1
     Section-label1 SECTION Priority-number?.
       Paragraph*1
       Label-name1. Statement*1.
       Label-name2. Statement*2.         %% paragraph to be moved
       Label-name3. Statement*3.         %% is Label-name2
       Paragraph*2
     Section*2
     BAR SECTION.
     BAR-PARAGRAPH.
       GOBACK.
     Section*3
     Section-label2 SECTION.             %% target SECTION; must be
       Paragraph*3                       %% subroutines section
     Section*4
   , Perform-labels                      %% variables containing lists
   , Perform-thru-labels                 %% of globally used par labels
   ) =
   move-paragraphs(
     shift-goto(
       Section*1
       Section-label1 SECTION Priority-number?.
         Paragraph*1
         Label-name1. Statement*1 PERFORM Label-name2.
         Label-name3. Statement*3.       %% added PERFORM statement
         Paragraph*2                     %% referencing Label-name2
       Section*2
       BAR SECTION.
       BAR-PARAGRAPH.
         GOBACK.
       Section*3
       Section-label2 SECTION.
         Paragraph*3
         Label-name2. Statement*2.     %% moved paragraph Label-name2
       Section*4
     , Label-name2                     %% these labels are used to
     , Label-name3                     %% transform each GO Label-name2
     )                                 %% into PERFORM Label-name2
   , Perform-labels                    %%      GO Label-name3
   , Perform-thru-labels               %% using shift-goto()
   )

[rule-shift-goto]
   shift-goto( GO Label-name2
   , Label-name2
   , Label-name3
   ) =
     PERFORM Label-name2
     GO Label-name3
```

Figure 5.15: Two ASF rewrite rules, taken from the Move-paragraphs transformation.

module in order to parse the corresponding equations. Compilation of the equations table to C code using the ASF compiler and then to a binary using the GNU C compiler took 25 seconds in total (compilation to C resulted in 18,000 lines of C code). Application of the compiled version to 80,000 LOC took 5 minutes, whereas the ASF interpreter took about 250 minutes for transforming the same code. The transformations that were performed on the source-base and in the case studies (Section 5.4) were done using the compiled version of our transformations.

Comments

The ASF+SDF Meta-Environment supports term rewriting with layout [46] when using the interpreter. This means that layout in sub-terms that are not rewritten are preserved. However, our transformations rewrote a lot of the code and therefore most of the layout was lost. For layout representing indentation this is no problem since a pretty-printer can format the code after the transformation. For layout representing comments it is a problem. Comments are lost if they are not represented in the grammar and rewrite rules. Moreover, representing comments in rewrite rules provides some control over the comments (e.g. adding comments, merging comments, moving comments). Therefore, we added productions for comments to the grammar and we modified the rewrite rules in order to preserve the comments in the source-base.

In the grammar, comments were added to the productions for statements, paragraphs, and sections. This meant that we had to add one non-terminal to these productions. In total, we added two productions for comments itself and we modified six existing productions. Using these productions, we were able to parse most of the comments in the source-base. The only comments that were lost were comments that appeared either inside the condition of an `IF` statement or before an `ELSE` keyword. For the code of the source-base, comments appearing in these places only consisted of dashes or blank lines and were therefore removed before we parsed the code. If we wish to keep all comments regardless their location, we can also use the scaffolding methods discussed in [174]. There, an approach is discussed for automatic generation of grammar extensions; this is done by extending the grammar until all code can be parsed. Our approach was sufficient for the code of the source-base.

We now discuss the limitations of using comments in the rewrite rules. Modification of the rewrite rules was slightly more complicated than modification of the grammar. We added about 370 comment variables to the rules, and we had to make decisions about what to do with the comments. Comments were attached to the next statement, paragraph or section. If a statement, paragraph or section was removed, its comments were also removed. However, we found some cases in the source code where comments belonged to the preceding statements. These were (incorrectly) attached to the next statement. In several transformation rules, it was not clear what should be done with the comments. For instance, when statements were merged, replaced or moved (note that the transformation pattern in Figure 5.15 has no comment variables, they were omitted). The example in Figure 5.16 is the transformation pattern which was also shown in Figure 5.10, with comment variables. In this pattern, the left-hand side has three variables which represent optional comments: `Comments?1`, `Comments?2`, and `Comments?3`. On the right-hand side,

```
Eliminate-GO(
  Comments?1                    Comments?1
  IF Condition                  IF Condition
    Statements1                   Statements1
    Comments?2                  ELSE
    GO Label                      Statements2
  ELSE                  ⟶        Statements3
    Statements2                 END-IF
  END-IF                        Comments?2
  Statements3                   GO Label
  Comments?3
  GO Label
)
```

Figure 5.16: Comment variables added to the Eliminate-GO pattern from Figure 5.10.

we must make a decision what to do with these comments. In the example we chose to keep `Comments?1` and `Comments?2`, but other options were to keep `Comments?3` instead of `Comments?2`, or to merge `Comments?2` and `Comments?3`. In practice, it turned out that `GO TO` statements were usually not preceded by comments. Another option is to place transformed code and its comments itself into a comment. For our transformations this is not useful because a great deal of code is transformed, and after a transformation it is difficult to determine where these comments should be placed.

In the source-base, there were 8,308 lines of comments in the procedure division; 2,789 of them contained alphanumeric or numeric characters, the rest were blank lines or filled with dashes or asterisks. Of these 2,789 lines of comments, we lost 130 lines of comments after transformation due to removed statements or paragraphs. This is about 5%. Since some transformations duplicate statements comments may also be duplicated. We measured the duplication and 10 lines of comments with alphanumeric or numeric characters were duplicated.

Summarising, it appears that it is not feasible to preserve and transform all comments as they were intended. Therefore, decisions about comments should be made on a per project basis in consultation with the maintainers or owners of the code.

5.4 Case studies: 5.2 million loc

One of the goals of the Restructuring project was to investigate how the transformations would perform on a large amount of source code that was not used when we calibrated them. Therefore, we did two case studies with the transformations on industrial Cobol systems. The first case study was done with 2.6 million LOC IBM Cobol and the second case study was done with 2.6 million LOC Micro Focus Cobol. The similar sizes were a coincidence but makes comparisons between the studies easier.

5.4.1 Case study I: IBM Cobol

The source code in the first case study was IBM Cobol and came from the same banking company as the code of the source-base. We had one large system of 2.6 million LOC in almost 1,000 programs. The program sizes ranged from about 40 LOC to 13,000 LOC. The number of statements per program ranged from 2 statements to 4,000 statements. In the whole system, there were about 400,000 statements.

The results of this study are summarised in Table 5.3. The transformation was done in 1 hour and 48 minutes. Afterwards, the number of statements was decreased by 3%. Almost 80% of the statements were extracted; this resembles the results of the source-base. More than 70% of the `GO TO` statements were removed, most of them were forward `GO TO` statements. The out-of-line `PERFORM` statements increased significantly, as they increase for each extracted paragraph. Similar to the source-base, the system initially contained no in-line `PERFORM` and `EVALUATE` statements so we could not measure the increase of these statements in terms of percentages.

The results of this case study indicate that our transformations can extract most of the statements and eliminate most of the `GO TO` statements in a different, larger system than we used as source-base. Many of the forward and backward `GO TO` statements, which are often most difficult to comprehend, were eliminated. Since we did not use this code to calibrate our transformations, we believe we can improve the results in this case study by analysing the resulting source code for new `GO TO` elimination transformation patterns. Next we will see how our transformations perform on code from a different company in a different Cobol dialect.

5.4.2 Case study II: Micro Focus Cobol

In the second case study we transformed several systems that were written in Micro Focus Cobol. Similar to the first case study, the total size was 2.6 million LOC but this was a coincidence. The source code consisted of almost 3,000 programs and the sizes of the individual programs ranged from 25 LOC to 8,000 LOC. The number of statements per program ranges from 10 to almost 3,400. In total, there were over 1.2 million statements. This significant higher number of statements compared to the first case study was because in the first case study a great deal of code was used for data declarations. More than 120,000 of the statements were `GO TO` statements, which meant that 1 out of 10 statements was a `GO TO` statement. This high ratio of `GO TO` statements compared to the first case study was due to the used programming style. Furthermore, implicit and explicit scope terminators were used interchangeably, causing more complexity.

The entire transformation took 5 hours and 23 minutes. Compared to the first case study, this longer duration is due to the higher number of statements and the higher ratio of `GO TO` statements. The results are shown in Table 5.4. Nearly 60% of the statements were extracted and 57% of the `GO TO` statements were removed, mainly forward `GO TO` statements. We compared these results with the results of the first case study and there was a significant lower percentage of extracted statements and eliminated `GO TO` statements. This was partly due to the relatively high number of exit and local `GO TO` statements; most of the remaining `GO TO` statements were exit or local ones. Before transformation, there were more exit `GO TO` statements than forward `GO TO` statements. Further-

Table 5.3: Results of case study I.

2.6 Million LOC IBM Cobol	Before	After	Changed
statements	405,073	393,106	- 3 %
`GO TO` statements	14,332	3,944	- 72 %
forward `GO TO` statements (not to `EXIT`)	10,891	1,531	- 86 %
exit `GO TO` statements	2,216	1,704	- 23 %
local `GO TO` statements	809	601	- 26 %
backward `GO TO` statements	416	108	- 74 %
in-line `PERFORM` statements	0	550	N.A.
out-of-line `PERFORM` statements	41,604	63,490	+ 53 %
`IF` statements	71,672	49,612	- 31 %
`EVALUATE` statements	0	5,517	N.A.
percentage of extracted statements	0 %	79 %	+ 79 %

Table 5.4: Results of case study II.

2.6 Million LOC MF Cobol	Before	After	Changed
statements	1,209,848	1,202,317	- 1 %
`GO TO` statements	122,934	52,674	- 57 %
forward `GO TO` statements (not to `EXIT`)	43,184	6,596	- 85 %
exit `GO TO` statements	45,276	20,035	- 56 %
local `GO TO` statements	14,649	20,146	+ 38 %
backward `GO TO` statements	19,825	5,897	- 70 %
in-line `PERFORM` statements	693	5,725	+ 726 %
out-of-line `PERFORM` statements	123,902	223,995	+ 80 %
`IF` statements	201,843	156,913	- 22 %
`EVALUATE` statements	11,738	19,498	+ 66 %
percentage of extracted statements	0 %	58 %	+ 58 %

```
1000.
  PERFORM OPEN-FILES.
1001-INIT.
  INITIALIZE PRODUCT-TABLE.
1003-ACCEPT.
  DISPLAY PRODUCT-SCREEN
  ACCEPT SEARCH-KEY.
  IF KEY-TYPE = "2"
    EVALUATE KEY-CODE
      WHEN ZERO
        GO TO 1099-EXIT
      WHEN 3
        INITIALIZE V-RECORD
        PERFORM PROCESS-PRODUCT
        MOVE V-NAME TO SEARCH-NAME
        GO TO 1001-INIT
    END-EVALUATE
  END-IF.
```

```
1000.
 PERFORM OPEN-FILES
 PERFORM 1001-INIT.

1003-ACCEPT.
 DISPLAY PRODUCT-SCREEN
 ACCEPT SEARCH-KEY
 IF KEY-TYPE = "2"
   EVALUATE KEY-CODE
     WHEN ZERO
       GO 1099-EXIT
     WHEN 3
       INITIALIZE V-RECORD
       PERFORM PROCESS-PRODUCT
       MOVE V-NAME TO SEARCH-NAME
       PERFORM 1001-INIT
       GO 1003-ACCEPT
   END-EVALUATE
 END-IF.
```

Figure 5.17: Transformation of a backward `GO TO` into a local `GO TO`.

more, this code initially had in-line `PERFORM` and `EVALUATE` statements. The number of `EVALUATE` statements increased significantly due to replacing nested `IF` statements by `EVALUATE` statements.

An interesting statistic is the increase of the number of local `GO TO` statements by 38%. We examined some of the source code to investigate this increase. It appeared that many of the backward `GO TO` statements were shifted into local `GO TO` statements. Although these `GO TO` statements were not eliminated, the modifiabilty was improved: complex backward `GO TO` statements were replaced by simpler local ones. Figure 5.17 illustrates this by a small code fragment before and after transformation. In the fragment, `1001-INIT` was extracted and therefore `GO TO 1001-INIT` was shifted to `PERFORM 1001-INIT GO 1003-ACCEPT`, which is a local `GO TO` statement.

The results of this case study show the performance of our transformations on large systems from a different company in a different Cobol dialect. Although the percentages of eliminated `GO TO` statements and extracted code are not as substantial as in the first case study, most of the forward and backward `GO TO` statements were removed. We believe that small changes to the transformations based on this source code can improve these results significantly.

5.5 Conclusions

We presented an approach to revitalize the modifiability of Cobol legacy assets using automatic restructuring transformations. The transformations consist of several rules that restructure source code while preserving the original appearance as much as possible. We presented a restructuring algorithm, the automatic transformations, and we showed that the transformations can easily be adapted to a system. We used a source-base of approximately 80,000 lines of code to calibrate the transformations. New transformation patterns were added to the existing ones without interference. Small changes in the rules improved the amount of transformed source code significantly. The resulting code consisted mainly

of small, extracted components without fallthrough that can be moved freely, and the code became modifiable using the created subroutines sections. In our source-base, 79% of the statements were extracted and placed in the subroutines sections and 82% of the `GO TO` statements were eliminated. Most of the remaining `GO TO` statements represented break or continue statements as used in other programming languages. It was not necessary to remove these `GO TO` statements for our purpose, which is making the code more modifiable and ready for proper evolution.

The approach was illustrated by applying the transformations to large real-life Cobol systems. We did two case studies which were in total 5.2 Million LOC but in two different dialects and from two different companies. One study concerned 2.6 Million LOC IBM Cobol and one study concerned 2.6 million LOC Micro Focus Cobol. In the first case study, 79% of the 405,073 statements were extracted and 72% of the 14,332 `GO TO` statements were removed. This transformation took 1 hour and 48 minutes. In the second case study, 58% of the 1.2 million statements were extracted and 57% of the 122,934 `GO TO` statements were removed. The second case study took 5 hours and 23 minutes for transforming the code. The case studies showed that the transformation is feasible on a large-scale, and illustrate the wide applicability of the approach.

The modifiability of real-life Cobol code is revitalized by applying the restructuring transformations. The code is more structured by extracting statements, removing several `GO TO` statements and shifting globally impacting `GO TO` statements to simpler local `GO TO` statements. This is achieved without introduction of synthetic variables or duplication and introduction of large amounts of code. New code can more easily be added to the created subroutines sections without the danger of disturbing the existing control-flow. Furthermore, due to the modularity of loosely coupled components, the code is better prepared for evolution like extraction, wrapping, integration, redevelopment, and replacement. In addition, as we illustrated in the Minefield project in Chapter 3, the restructuring transformations can help to fight evolved, error-prone code.

Acknowledgements We thank all reviewers for their comments. The research in this chapter was supported by the Software Improvement Group Amsterdam [184], who also provided the source code for the case studies. We are very grateful to Steven Klusener for his help. He also did most of the work on developing the grammars that we used in the case studies. We also thank Mark van den Brand, Ralf Lämmel, Chris Verhoef and Jurgen Vinju for their ideas, comments and support.

Road map We discussed large-scale error analysis in Chapter 3, large-scale local modifications in Chapter 4, and large-scale restructuring in the current chapter. Now we move on to another level, that is, the analysis and transformation of the transformations themselves. In Chapter 6, we presents several techniques to exercise control over large-scale maintenance changes. We study the restructuring transformations from the current chapter and the large-scale changes from the Btrieve project from Chapter 4, and we propose an approach to check such changes in a cost-effective and practical way.

Chapter 6

Towards lightweight checks for mass maintenance transformations

In this chapter, we propose a lightweight, practical approach to check mass maintenance transformations. We present checks for both transformation tools and transformed source code, and illustrate them using examples of real-world transformations. Our approach is not a fully fledged, formal one but provides circumstantial evidence for transformation correctness. We applied the approach to the mass update project from Chapter 4 and to the restructuring effort from Chapter 5. This chapter is based on: N. Veerman. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*, 57(2):129–163, 2005 [195].

6.1 Introduction

Transformations are very important for software engineering. Compilers, code generation, code analysis, automated modifications and refactoring are only a few examples of commonly known transformations. The use of transformations has gradually increased in the field of software maintenance. For example, code analysis tools have become more and more indispensable to maintain large systems, and large-scale modifications, such as Euro conversions, Y2K repairs or database migrations, have called for automated maintenance transformations [199].

The increasing interest in automated maintenance has heightened the need for control over automatic maintenance transformations. For instance, automated changes on business-critical systems should not jeopardize the operations of a company. Such changes usually consist of several complex transformations, affecting millions of lines of code. It is not very practical to inspect the result of a mass update by hand. An alternative, extensive testing of the transformed code, is expensive and not always feasible in practice. Massive automated changes are often carried out by software renovation companies, and

they usually do not possess the required hard- and/or software to, for instance, compile and test a mainframe system. Although the owner of the system has a suitable compiler, mainframes are delivered with a fixed calculation capacity. The employment of this capacity for finding errors in (automated) changes is expensive (see [78] for figures on costs for compiling large systems), so possible errors should be detected at an early stage. Therefore, additional ways are needed to improve control over such transformations.

Some people think that it is feasible to prove the correctness of a mass maintenance transformation on millions of lines of industrial code in advance. However, correctness proofs are time-consuming and often impractical for real-life mass maintenance transformations. We will explain this with some examples.

In order to deploy correctness proofs, one needs the semantics of the used programming language. These have not been defined for every programming language; in particular, legacy languages like Cobol have no uniform semantics. There are many different dialects, compilers, compiler flags and operating systems, and thus there is no single semantics for these languages. It is important to recognise this issue, and it is discussed in [130, 131]. As it is not commonly known that there is no single semantics, we show some of the examples from [130] here. Consider the following Cobol fragment, taken from [130, p83]:

```
PIC A X(5) RIGHT JUSTIFIED VALUE 'IEEE'.
DISPLAY A.
```

This fragment illustrates differences between two compilers. The OS/VS Cobol compiler prints the expected result, namely ' IEEE', which is right justified, whereas the Cobol/370 compiler displays the output 'IEEE ' with a trailing space, which is *left* justified. The second example illustrates differences within a single compiler, and is also taken from [130, p83]:

```
01 A PIC 9999999.
MOVE ALL '123' TO A.
DISPLAY A.
```

Depending on the compiler flags, this code displays the number '3123123' or the entirely different number '1231231'. There are many more similar examples, for instance, we showed in the Minefield project in Chapter 3 that the precise semantics of the behaviour of procedure calls in Cobol is compiler-dependent. The large amount of variation of semantics for languages and technologies make it unattractive to deploy formal proofs for mass maintenance transformations.

On the other hand, one could capture a subset of the semantics for a specific maintenance project with a specific Cobol dialect, compiler and operating system, i.e. relevant for a particular modification [14]. However, using a formal approach can still be an expensive process and prone to errors, and as soon as something changes (e.g. compiler version or operating system version) a different semantics needs to be obtained. Hence, it is not surprising that the ROI on correctness proofs is negative [107, p108]. Moreover, in mass maintenance projects it is not always the goal to preserve the functionality of a system; hence, such modifications are not always semantics and correctness preserving, as we explained in Chapter 4. Also, in addition to proving the transformations correct,

one would also have to consider the rest of the transformation process, for instance the preprocessor, parser and pretty printer and so on. For these reasons, a correctness proof for mass maintenance transformations on legacy systems is too expensive and not practical for industrial projects, and we need to search for other ways to make sure the updated systems behave as expected.

The aim of this chapter is to provide a lightweight approach to check large-scale maintenance transformations. The approach is in between extensive testing of the modified system and formal techniques. We present a range of checks to identify errors in transformations quickly and at low cost, and we applied these checks to two mass maintenance projects: the Btrieve project from Chapter 4 and the Restructuring project from Chapter 5.

Related work Related work in the field of program transformations can be found, for example, in Partsch [159], where transformational programming is described. Their primary goal is that of general support for program modifications. This includes the generation of programs from formal specifications, adaptation of programs to particular environments, and program descriptions. However, no methods for checking transformations is given. Taxonomies of program transformations are described in [66], and in [207] program transformations are classified as *translations*, i.e. the source and target language are different, or as *rephrasings*, i.e. source and target language are the same. Both types of program transformations can be found in the area of software maintenance automation.

An automatable methodology for formally proving the correctness of transformation systems is described, constructed and applied in [214]. This can only be applied if the semantics of the programming language is available, which is not the case for Cobol. Also, the use of formal proofs in automated maintenance is not always practical, as we argued above. In [29] a transformation system for Cobol is discussed. They state that their transformations are often little more than a recasting of an algebraic law, which makes it plausible that the transformed code behaves as expected. This is not the case for our transformations: these can be complex and highly customisable.

In [121] a complex migration from PL/IX to C++ was carried out. The target language was generated by semi-automatic transformations. The resulting code was verified by passing it through a set of tests, but they do not report about verifying or checking the transformations itself, which is desired in such projects. Another migration is presented in [210], where assembler code is converted to equivalent C code using the FermaT system. The transformation engine has a library of proven transformations to preserve or refine the semantics of a program. They report that resulting C code compiled without warnings or errors, but they do not mention whether the results were tested and taken into production, or any other way to check the results. In a more recent article from the same author [211], another assembler to C migration is discussed. In the article the resulting code was reviewed, examined and undergoing final testing by the customer. Hence, possible errors in the transformations have to be discovered by the customer. Our approach aims at detecting errors as early as possible, and especially before delivering code to a customer.

Starting point and outline Our starting point is two automated mass maintenance projects: the Btrieve project from Chapter 4 and the Restructuring project from Chapter 5.

In both projects, large amounts of industrial Cobol source code was transformed with automatic tools. The Btrieve project involved a database upgrade for an entire software portfolio, the Restructuring project involved code restructuring for Cobol. We applied our lightweight checks to the projects to evaluate our method.

This chaper has been organised as follows. In Section 6.2, we briefly review the Btrieve project and the Restructuring project, provide some examples, and give an overview of our lightweight checks for mass maintenance transformations. In Section 6.3 and Section 6.4, our checks are described. Then in Section 6.5, we discuss the application of our approach to the Btrieve project and the Restructuring project, and in Section 6.6 we briefly analyse how much time was needed to implement and apply our approach. In Section 6.7 we summarise and conclude.

6.2 The two mass maintenance projects

In this section we briefly review the two industrial mass maintenance projects from the two previous chapters: the Btrieve project and the Restructuring project. The two projects are different, in a way that the first one involved small, one-line changes in several systems, whereas the second one involved changes to the structure of entire programs. We illustrate the projects with small transformation examples and argue for ways to increase control on such projects. Then we give an overview of our approach for checking mass maintenance transformations. A detailed treatment of our approach will be given in Section 6.3 and Section 6.4.

6.2.1 Project I: the Btrieve project (Chapter 4)

The first project involved an upgrade of Cobol systems to deal with an upgrade of the database system, which was due to scalability and performance issues. In order to cope with constraints that were imposed by the new database version, the existing Cobol systems that accessed the database had to be altered. The modification was concerned with five different database operations that had to be examined and possibly modified; this meant that we analysed dataflow and modified variables that were used in the database calls. The entire Cobol portfolio had about 50 thousand database calls, which were spread over 4 million lines of code. This included almost 3000 programs and nearly 20 thousand include files (copybooks). The full treatment of the Btrieve project can be found in Chapter 4.

We show a simplified example in Figure 6.1, which illustrates one of the changes that were made. In the code on the left-hand side, the `call` statement calls the database `BT` and opens the file specified in `key-buf`. The open mode is specified by the last argument, `key-1`. The variables `data-buf` and `data-bufl` can be used transfer data to and from the database, and `key-buf` returns the positioning information about the file. Then on the right-hand side, the `key-1` variable is transformed when its value is not equal to zero. It is replaced by `key-0` with value 0, and this new variable is added to the data declarations if it is not yet declared.

Similar to this example, four other database operations had to be modified if the value of some of the call arguments was not in accordance with the constraints of the new

```
..
DATA DIVISION.
01 ...

..

PROCEDURE DIVISION.
..
call BT using b-cre,b-status,pos-block,
      data-buf,data-bufl,key-buf,key-1.
..
```

Before transformation

```
..
DATA DIVISION.
01 ..
01 key-0 PIC 9(4) COMP-5 VALUE 0.
..

PROCEDURE DIVISION.
..
call BT using b-cre,b-status,pos-block,
      data-buf,data-bufl,key-buf,key-0.
..
```

After transformation

Figure 6.1: Code sample from the Btrieve project.

database version, and fresh variables had to be declared. This transformation project involved analysis of database calls and data declarations in the program and in the include files, as well as tracking variable values throughout programs. To have confidence in the modifications at low-cost, we need ways to detect errors prior to compiling and testing the updated code.

6.2.2 Project II: the Restructuring project (Chapter 5)

The second project involved restructuring of Cobol code to improve the modifiability. The main goal was to componentise monolithic code by restructuring complex logic, thereby enabling the system for new changes. About 25 different transformations were combined in an algorithm to restructure legacy Cobol. Each transformation dealt with a particular language construct; for instance, complex branching statements were restructured, goto statements were removed, imperative code was transformed into subroutines and dead code was removed. During the project, the transformations were first calibrated with a system of 80,000 lines of code and then applied to over 5 million lines of code; in this chapter we apply our checks to the 80,000 lines of code system that was used for calibration. The full treatment of the project can be found in Chapter 5, the used restructuring algorithm originates from [171].

The example in Figure 6.2 shows a code snippet from one of the programs before and after transformation. Here, several transformations are applied to the code fragment on the left-hand side, and the result is shown on the right-hand side. These transformations extract labelled statements into called procedures (labels `1001`, `1003` and `1005` are turned into subroutines), eliminate goto statements, and normalise code. In this example, several transformations were applied to the code fragment, some of them more than once. Although the code is more structured after the transformations, it is difficult to see whether the functionality remained the same. Tests can be used to to check the behaviour of the code, but additional methods are desired.

```
1001.
  IF OP-VL1=150 GO TO 1005.
  MOVE 0 TO VE-VL1 (OP-VL1).
  SET OPF-VL2 TO 1.
1003.
  IF OP-VL2=100
    SET OP-VL1 UP BY 1
    GO TO 1001.
  MOVE 0 TO GL-VL1 (OP-VL1,OP-VL2).
  SET OP-VL2 UP BY 1.
  GO TO 1003.
1005.
  MOVE 0 TO REVS-GL1 (OP-GL1).
  SET OP-GL1 UP BY 1.
```

Before transformation

```
MAIN.
  PERFORM 1001
  PERFORM 1005
  GOBACK.

1001.
  PERFORM TEST BEFORE UNTIL OP-VL1=150
    MOVE 0 TO VE-VL1 (OP-VL1)
    SET OP-VL2 TO 1
    PERFORM 1003
  END-PERFORM.
1003.
  PERFORM TEST BEFORE UNTIL OP-VL2=100
    MOVE 0 TO GL-VL1 (OP-VL1,OP-VL2)
    SET OP-VL2 UP BY 1
  END-PERFORM
  SET OP-VL1 UP BY 1.
1005.
  MOVE 0 TO REVS-GL1 (OP-GL1)
  SET OP-GL1 UP BY 1.
```

After transformation

Figure 6.2: Code sample from the Restructuring project.

6.2.3 Mass maintenance checking

The examples from the projects illustrate the need for more control over mass maintenance transformations. If such transformations have been applied to millions of lines of source code, (semi-)automatic checking is very useful. We propose a lightweight approach consisting of several checks. These checks can be used to detect errors in transformations but they do not guarantee the absence of errors; we say that they provide *circumstantial evidence* for transformation correctness. Moreover, a successfully applied check increases the confidence in the transformations. The checks have been divided into the following two categories:

- Transformation rule checks: these checks involve the actual transformation rules. The transformation rules in the Btrieve project and Restructuring project consist of a left-hand side pattern and a right-hand side pattern, and may have conditions. The advantage of these checks is that they can be applied statically, thus without applying them to the program code.

- Program code checks: these checks are performed on the program code before and after transformations. The advantage of these checks is that the entire transformation process is taken into account, since these checks are applied on the original code and final code. Thus the pre- and postprocessing of the code are also checked.

We discuss both automatic and semi-automatic checks in the subsequent sections. We summarised them in Table 6.1.

Table 6.1: Overview of the lightweight checks.

Transformation rule checks	Program code checks
Control-Flow Invariance (CFI)	Frequency Characteristics (FC)
Variable Consistency (VC)	Compilation & Regression Tests (CRT)
Grammar-based Testcase Generation (GTG	

The first transformation rule check, Control-Flow Invariance (CFI), aims at checking whether control-flow of the input pattern is equal to the control-flow of the output pattern using a bisimulation equivalence relation. This is useful as a lightweight check on the behaviour of the input and output transformation pattern. The second check, Variable Consistency (VC), aims at detecting errors in the variables of the transformation rules by reporting modified and/or removed variables. The third check, Grammar-based Testcase Generation (GTG), uses the grammar of the programming language and the transformation rules to generate testcases for the rules, which can also be used to find errors.

The first program code check, Frequency Characteristics (FC), can be used to identify errors in transformations by looking at certain properties of the program code. If all of the checks succeed then Compilation & Regression Tests (CRT) is a final check.

We applied most of these checks to transformations and program code from the Btrieve project and the Restructuring project, and the results are presented in Section 6.5. We want to emphasise that our checks do not prove correctness of transformations, but the checks are a lightweight approach to detect errors. The transformation rule checks are discussed in Section 6.3 and the program code checks are discussed in Section 6.4.

6.3 Transformation rule checks

In both projects, we used the ASF+SDF Meta-Environment [45, 112], which we discussed extensively in Chapter 2. We briefly consider those aspects of the transformation system that are relevant for this chapter. Then we present the transformation rule checks Control-Flow Invariance (CFI), Variable Consistency (VC) and Grammar-based Testcase Generation (GTG).

As we mentioned above, the transformations of the projects were implemented in the ASF+SDF Meta-Environment. For our transformations, SDF was used to parse source code and build a parse tree. A parse tree represents the source code using *terminals* and *non-terminals* from the grammar. Then code patterns in the tree were rewritten with (parsed) ASF rewrite rules; a rule matches if the left-hand side of rule matches and the conditions are successfully evaluated. In this chapter, we will denote a rewrite/transformation rule as shown in Figure 6.3. An example transformation rule from the Restructuring project is shown in Figure 6.4. The rule is part of the Eliminate-empty-IF transformation, which restructures if statements. If there are no statements in the if-branch, the condition is inverted and the statements from the else-branch are placed in the if-branch. In this example, `empty` and `eliminate-empty-if` are function symbols, and `Condition`, `Statements1` and `Statements2` are abstract syntax variables

```
conditions
⟹
left-hand side                    ⟶  right-hand side
```

Figure 6.3: Notation for a transformation rule.

```
empty( Statements1 ) == true
⟹
eliminate-empty-if(
  IF Condition THEN               ⟶  IF NOT( Condition ) THEN
    Statements1                         Statements2
  ELSE                                END-IF
    Statements2
  END-IF
)
```

Figure 6.4: Transformation rule for restructuring an if statement.

representing non-terminals of the type condition and statements. The `IF`, `THEN`, `NOT`, `ELSE` and `END-IF` are concrete syntax representing terminals of the Cobol syntax. In the following sections, we present a number of transformation rules and discuss how we perform checks on them to detect possible errors.

6.3.1 Control-Flow Invariance

Our first check on transformation rules concerns the Control-Flow Invariance of the input and output pattern. This check can be used for transformations which are supposed to preserve the control-flow, such as restructuring and refactoring. We compare the statements on the left-hand side with the statements on the right-hand side using a bisimulation equivalence relation. This verifies if on both sides the same statements can be executed at the same moment. If this fails, there can be an error in the transformation rule. For instance, there can be a typographical error, or an incorrect transformation. The input and output patterns are converted into process graphs representing the control-flow, which can then be fed to a bisimulation checker. The bisimulation checker will either fail or succeed. If it fails, it reports where an error can be found. We experienced that comparing the control-flow graphs of the input and the output pattern is an effective way to (automatically) check a transformation rule.

We will discuss briefly what bisimulation equivalence means and which type of bisimulation equivalence we use. Then we explain how we translate input and output patterns to control-flow graphs using a tool we developed. Finally, we describe how we compare the control-flow graphs of the patterns using a bisimulation checker.

Bisimulation equivalence Two process graphs are bisimilar if they can execute exactly the same sequence of actions, and have the same branching structure [11, 82]. The branching structure is similar if, in both process graphs, the same choices can be made at the same moments. A special action is the *silent step*, which can be used to abstract from internal actions. There are several types of bisimulation equivalence relations [12, 11, 91]. We briefly discuss *strong* and *branching* bisimulation. Strong bisimulation treats a silent step as a normal action. If a silent step is possible at a certain moment in a process graph, it should be possible in the other process graph at the corresponding moment. Branching bisimulation, on the other hand, treats silent steps as *truly* silent steps. This means that if a state in one process graph can take a silent step, then this need not be the case for the corresponding state in the other process graph. We use branching bisimulation for transformation rules that eliminate goto statements, which can be represented by silent steps. In such rules, silent steps will appear only in the process graph of the left-hand side of the transformation pattern, because the goto statement is removed on the right-hand side.

One could also use *trace equivalence* to compare control-flow graphs, which is a less strict equivalence relation than bisimulation equivalence. Trace equivalence means that the same execution traces can be found in both graphs, i.e. the same sequence of actions can be performed. However, that form of equivalence does not consider the moment an action can be taken, and two graphs can be trace equivalent while not bisimilar. Hence, for our approach, bisimulation equivalence can detect more potential errors.

Translation of patterns to graphs Before we can compare the input and output pattern of a transformation rule using a bisimulation checker, we need to translate them to process graphs representing the control-flow. We developed a tool that converts a pattern into a graph. To represent these graphs, we used Labelled Transition Systems (LTS). An LTS consists of a collection of states and a collection of transitions between them. Our tool converts an input or output pattern into an LTS in Aldebaran [80] format, consisting of a descriptor and edges. The descriptor has the following structure:

```
des (<start-state>,<number-of-transitions>,<number-of-states>)
```

The edges have the following structure:

```
(<from-state>,<transition-name>,<to-state>)
```

Figure 6.5 displays an example, where an input transformation pattern is on the left-hand side and its LTS on the right-hand side. The corresponding control-flow graph is illustrated in Figure 6.6.

We see that the LTS has seven states (0 thru 6) and seven transitions. Each statement variable (`Statements1`,...,`Statements4`) has been translated into an edge and there is an edge for the `Condition` and there is one for `NOT Condition`. The goto statement has been translated into a silent step, which is denoted by an '`i`' (internal step) in Aldebaran format. The `Label` is not represented in the graph because a label itself cannot be executed; it is only a reference point. Note that we encode the behaviour of the control statements if and goto in the translation tool, but we can abstract from the rest of the language since it is not relevant for the transformation.

Transformation pattern	**Labelled Transition System**
Label.	des (0, 7, 7)
Statements1	(0, Statements1, 1)
IF Condition	(1, Condition, 2)
Statements2	(1, NOT Condition, 3)
GO TO Label	(2, Statements2, 4)
ELSE	(4, i, 0)
Statements3	(3, Statements3, 5)
END-IF	(5, Statements4, 6)
Statements4.	

Figure 6.5: A transformation pattern and its Labelled Transition System.

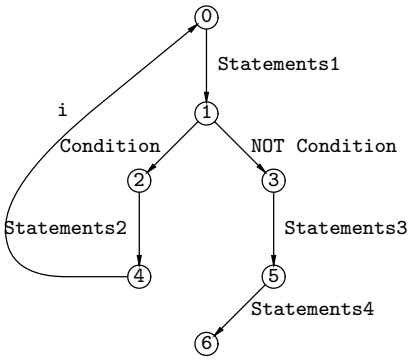

Figure 6.6: The control-flow graph of the transformation pattern from Figure 6.5.

```
⟹
remove-dead-code(
  Statements1                    ⟶  Statements1
  GO TO Label                       GO TO Label
  Statements2
)
```

Figure 6.7: Transformation rule for removing dead code.

Comparing the graphs To compare Labelled Transition Systems using branching bisimulation, we use Aldebaran. Aldebaran is Unix-based and part of the Caesar-Aldebaran Development Package [80], and allows the minimisation and comparison of LTSes with respect to various equivalence and preorder relations. Branching bisimulation equivalence, as well as other relations, can be used for comparison. In our approach, a transformation pattern is converted into an input and an output LTS, and these two LTSes are then compared using Aldebaran. We illustrate this with an example.

In Figure 6.7 a simple transformation rule for removing dead code is shown. On the left-hand side `Statements2` will never be executed because it is preceded by an unconditional jump, and can therefore be removed. The control-flow graphs are shown in Figure 6.9; corresponding states are connected by dashed lines. We can see in the graphs that `Statements2` cannot be reached from state 0 and it can safely be removed. The input pattern and output pattern are converted into LTSes (`.aut` files represent Aldebaran files). These are shown in Figure 6.8. The LTSes are then fed to Aldebaran, which returns `TRUE` because they are branching bisimilar (`-pequ` means compare using branching bisimulation):

```
$ aldebaran -pequ deadcode.in.aut deadcode.out.aut
TRUE
$
```

Summary We believe the CFI check is a quick and effective way to detect errors in transformation rules. A drawback of our translation tool is that it cannot deal with rule conditions that affect the control-flow. Note that the CFI check can also be applied to program code, thereby translating statements to edges, but we have not investigated this. In Section 6.5 we will discuss how we found using CFI an incorrect goto-elimination rule in the Restructuring project.

6.3.2 Variable Consistency

Our second transformation rule check concerns the Variable Consistency of a rule. In a transformation rule, it may or may not be the purpose to modify or remove a variable. To detect unwanted modification or removal of variables, we defined a consistency check. This check is the only check which takes conditions of the rules into account. We will start by giving some examples of transformations that may not be correct due to either a typographical or a transformation error. Then we describe the consistency check.

Input LTS: `deadcode.in.aut`	**Output LTS:** `deadcode.out.aut`

`des(0, 3, 5)`
`(0, Statements1, 1)`
`(1, GO TO Label, 2)`
`(3, Statements2, 4)`
	`des(0, 2, 3)`
	`(0, Statements1, 1)`
	`(1, GO TO Label, 2)`

Figure 6.8: LTSes for the transformation rule from Figure 6.7.

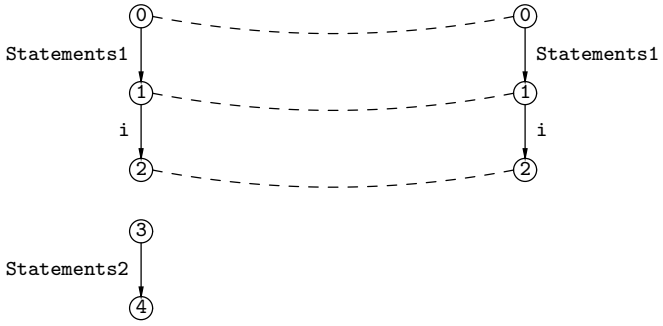

Figure 6.9: Bisimulation equivalence of the transformation rule from Figure 6.7.

The transformation in Figure 6.10 merges procedures if they can only be executed consecutively, i.e. executing the first implies executing the second procedure and the second cannot be executed without first executing the first one. A condition should make sure that the second label is not referenced, and thus can be removed. However, there is a typographical error in this transformation rule. Instead of the second procedure (`Label2`), the first procedure (`Label1`) is in the condition. As a consequence, `Label2` is thrown away without checking whether it is referenced.

We would like to detect such potential errors automatically by placing a constraint on the transformation rules. According to [22, 114], a rewrite rule must comply with two constraints. First, the left-hand side of a rule is not a variable, and second, the variables on the right-hand side are contained in the left-hand side. However, as also noted in [114, p 101], for Conditional Term Rewriting Systems it would make good sense to lift the second constraint. A variable on the right-hand side can be defined in a condition. And the first constraint may also be a hindrance to the rewrite rules used in mass maintenance transformations. Therefore, we propose a different constraint on transformation rules:

- *Variables appearing on only one side of a transformation rule should be contained in a condition.*

A variable that appears only on the right-hand side of a rule would lead to uncontrolled behaviour [22, p36], as the particular variable is not bound. A variable that appears only on the left-hand side of a rule is considered to be erased. Such a rule is called *erasing* [22, p39]. It may or may not be the intention to erase a variable in a rule. Hence, a rule that

```
occurs-in( Label1, Referenced-labels ) == false
⟹
merge-procedure(
  Label1.                          ⟶  Label1.
    Statements1.                         Statements1
  Label2.                                Statements2.
    Statements2.
, Referenced-labels
)
```

Figure 6.10: Erroneous transformation for merging procedures.

```
Label3 == Label1
Label4 == Label2
⟹
swap-labels(
  Label1.                          ⟶  Label3.
    Statements1.                         Statements1.
  Label2.                                Label4.
    Statements2.                         Statements2.
)
```

Figure 6.11: Transformation example with two errors, cancelling each other out.

does not comply with our constraint does not have to be erroneous. But using our VC constraint, we want to reduce the search space when searching for erroneous rules. The VC constraint detects the potential errors in the transformation we showed above, and can detect several others in transformation rules.

However, there are still cases that will not be detected, see an example in Figure 6.11. That transformation rule swaps two labels, and, for some reason, two intermediate labels are used to store and replace the value of the labels. However, the assignments in the conditions have been swapped themselves. Hence, this transformation contains two errors. The VC constraint we impose will not detect this transformation error because all variables that are on only one side are contained in the conditions. On the other hand, these are probably typographical errors which are not detected by the check because they cancel each other out. Similar behaviour can be observed in a spell checker. For instance, if the word 'compiled' is spelled as 'complied', two letters are swapped, but the spell checker will not report this error because 'complied' is a valid word.

Summary Taken together, we believe our VC check can detect possible errors in transformation rules. It is difficult to check rule conditions in a lightweight way, and this check is the only transformation rule check we have that takes such conditions into account. Therefore, this check can be used in addition to the two other transformation rule checks, the CFI check and the GTG check. In Section 6.5 we apply the VC check to the transformation rules from the Btrieve project and the Restructuring project.

6.3.3 Grammar-based Testcase Generation

Our last check for transformation rules involves Grammar-based Testcase Generation. The check originates from the field of automatic generation of test data. A comprehensive overview of this subject is given in [103]. That paper reports on syntax-based testing in several areas, especially in compiler testing. Since our transformations are grammar-based it would be natural to generate tests in a similar way. However, there are several difficulties when using grammars for generating test data. The most prominent problem is probably to generate a set that has relevant tests and that is not too large. To deal with these issues, work is done to enhance a grammar [145] and to control the generation process [128]. As far as we know, generation of testcases for software maintenance automation is still an unexplored area.

Transformation rule coverage

Consider the rule for eliminating a goto statement in Figure 6.12. The rule transforms a pattern with a local jump into a while-like loop. In this rule, the variables indicated with `Statements` denote zero or more statements; hence, the appearance of statements in those places is optional. Input code (e.g. Cobol) is affected by the rule if it matches the left-hand side, and code is not transformed if not. For instance, in Cobol, the if statement can have an optional `THEN` keyword (for the IBM VS Cobol II grammar we refer to [129]). This optional keyword is not represented in the pattern so if in Cobol code an if statement is encountered with the `THEN` keyword, the transformation pattern will not match. In this particular rule, it was the intention to have this keyword omitted, but that cannot be seen from the transformation rule and tests are needed to document this behaviour.

There are many more cases for this rule which nearly match, and it would be useful to let them document the transformation rule, as well as cases that do match. If we annotate the left-hand side of the rule from Figure 6.12 with information about optional terminals and non-terminals according to the grammar, we obtain a rule as shown in Figure 6.13. Terminals are depicted with quotes. If in the source code any of the square brackets is 'filled' with code, the transformation rule will not match. Hence, when one is developing, maintaining and testing grammar-based transformation rules, it is useful to know what the production rules in the grammar look like and to what extent these are covered by the transformation rules.

Automatic testcase generation

We have developed an approach to generate a set of testcases automatically for a transformation rule. The set is derived from the grammar and documents the coverage of a rule. Cases are classified as either *matching* or *non-matching*. A matching case is defined as a code fragment on which the transformation rule can make a change. A non-matching case is defined as a code fragment that is not affected by the transformation rule. We propose the following algorithm for generating these cases:

- For a set of matching cases: Generate a case for each optional (non-)terminal on the left-hand side of the transformation rule and a case without any optionals.

```
⟹
eliminate-goto(
 Label.                          ⟶  Label.
   Statements1                        Statements1
   IF Condition                       PERFORM TEST BEFORE UNTIL
     Statements2                           NOT (Condition)
     GO Label                         Statements2
   ELSE                               Statements1
     Statements3                      END-PERFORM
   END-IF                             Statements3
   Statements4.                       Statements4.
)
```

Figure 6.12: Goto-elimination transformation rule.

```
Label.
  Statements1
  IF Condition ["THEN"]
    Statements2
    GO ["TO"] Label [In-label]
    [Statements]
    [Statement-non-closed]
  ELSE
    Statements3
    [Statement-non-closed]
  END-IF
  Statements4
  [Statement-non-closed].
```

Figure 6.13: Left-hand side of the rule from Figure 6.12 with optional (non-)terminals.

- For a set of non-matching cases: Generate one case for each (non-)terminal that is present in the grammar productions of the left-hand side of the transformation rule but not in the transformation rule itself.

With this algorithm, the number of generated cases will not explode since we do not generate each possible combination for matching and non-matching cases. Moreover, we keep a generated code pattern as small as possible by leaving out parts of the pattern that are not relevant to the particular case. For example, if matching cases are generated for the rule from Figure 6.12 and the case for the non-terminal `Statements1` is generated, the other optional (non-)terminals remain empty. This way, the number of matching cases will be at most the number of variables in the pattern that may be empty plus one, while the number of non-matching cases may become larger, depending on the transformation pattern and the grammar. In addition, non-matching cases can be annotated with the terminal or non-terminal that causes non-matching behaviour, thereby automatically documenting the case.

We demonstrate our approach on the transformation pattern from Figure 6.12. Recall that our transformations were implemented in the ASF+SDF Meta-Environment. In that environment, transformation rules are first parsed itself before they can be applied to code; hence, a parse tree of the rules can quickly be obtained. We reused the parsed rules as a starting point for generating testcases, which simplified the analysis of transformation rules. The complete generation process consists of three steps:

1. Extract syntax information from the parsed transformation rule;

2. Build matching and non-matching patterns with the syntax information;

3. Generate cases from the patterns.

We illustrate this process with the example transformation rule from Figure 6.12.

Step 1 Extract information In the first step we extract syntax information from the parsed transformation rule. This means that we retrieve information about the (non-)terminals in the left-hand side pattern. This way, we can detect (non-)terminals that are present in the grammar productions but not in the transformation pattern. Information, as shown in Figure 6.13, is then revealed which is not shown in the transformation rule.

Step 2 Build patterns In the second step our algorithm comes into play. Based on the information from Step 1, we build matching and non-matching patterns using the retrieved terminals and non-terminals. These patterns are not yet real code because they still contain non-terminals. In Figure 6.14 a matching pattern is shown; in this case the non-terminal `Statements` is left in the else-branch. In total, 5 matching patterns are built for this rule: one for each variable in the pattern plus one empty variant without optionals.

A non-matching pattern is shown in Figure 6.15; matching is prevented by the non-terminal `Statement-non-closed`. In total, 7 non-matching patterns are built: one for each (non-)terminal that is not represented in the pattern.

```
Label.
  IF Condition
    GO Label
  ELSE
    Statements
  END-IF.
```

Figure 6.14: Matching pattern for the rule from Figure 6.12.

```
Label.
  IF Condition
    GO Label
  ELSE
    Statement-non-closed
  END-IF.
```

Figure 6.15: Non-matching pattern for the rule from Figure 6.12.

```
COBOL-WORD.
  IF ALPH-USERDEF-WORD
    GO COBOL-WORD
  ELSE
    ACCEPT ALPH-USERDEF-WORD
  END-IF.
```

Figure 6.16: Generated case for the matching pattern from Figure 6.14.

```
COBOL-WORD.
  IF ALPH-USERDEF-WORD
    GO COBOL-WORD
  ELSE
    ADD ALPH-USERDEF-WORD TO ALPH-USERDEF-WORD
      SIZE ERROR ACCEPT ALPH-USERDEF-WORD
  END-IF.
```

Figure 6.17: Generated case for the non-matching pattern from Figure 6.15.

Step 3 Generate cases The last step is the actual generation of code using the grammar. For each non-terminal for the patterns from Step 2 we generate code. It is unimportant what code is generated, as long as it is syntactically correct. All lexical sorts are initialised with a descriptive string for the sort, and generated cases are kept as small as possible. When a production has several alternatives we take the first alternative.

In Figure 6.16 the generated code for the matching pattern from Figure 6.14 is shown. In the grammar, the smallest production for a label is a cobol-word, so generating a label yields the string `COBOL-WORD`. Then the smallest production for a condition is an alphabetic-user-defined-word; hence, the condition variable is replaced by the string `ALPH-USERDEF-WORD`. Then statements reduces to a single statement, and the first alternative for a statement is the accept statement. An accept statement, which accepts an identifier, is generated. The identifier yields an alphabetic-user-defined-word.

In Figure 6.17, the generated code for the non-matching pattern from Figure 6.15 is shown. In addition to productions for labels and statements, a non-closed add statement with a `SIZE ERROR` option is generated. Inside the non-closed add statement, an accept statement is generated, as this is the first alternative for a statement.

We illustrated our generation process by a transformation rule with a matching a non-matching case. This entire process was fully automatic. Since the generated cases do not take rule conditions into account, the cases can be transformed by the transformation rule and then automatically re-classified as matching or non-matching.

Generated cases can serve as unittests and documentation, which are updated automatically if something changes. Changes to the grammar can automatically be carried through to the testcases, thereby indicating that perhaps the transformation also has to be updated. On the other hand, an updated transformation rule can result in different behaviour on the generated cases of the previous version of the transformation, so (unwanted) changes can be detected automatically.

We were able to quickly extract syntax information from the transformation rules. Although it may be due to the used technology (the ASF+SDF Meta-Environment) that we can do it quickly, this approach can probably be applied to other grammar-based transformation technologies to some extent as well.

Summary The GTG check is a useful way to generate testcases for transformation rules automatically based on the grammar. We have shown how we generate matching and non-matching cases for a given transformation rule. We believe that further investigation and incorporation of similar techniques in transformation systems would be useful for software maintenance transformations. In Section 6.5, we apply this check to the transformations from the Btrieve project and the Restructuring project.

6.4 Program code checks

Program code checks are applied to the source code of the application that is being transformed. In the previous section, we discussed CFI, VC and GTG for checking the transformation rules themselves. These checks cannot detect certain types of errors and there-

fore we defined additional checks. Checks that are applied to program code can also detect errors that are made elsewhere in the transformation process, for example in a pre- or postprocessing phases of the code. We propose two checks: Frequency Characteristics (FC), which uses properties of the program code to detect errors, and Compilation & Regression Testing (CRT), which will usually be performed at the customer.

6.4.1 Frequency Characteristics

The FC check uses an analysis of the frequencies of elements in the program code to detect errors in transformations. For example, the frequency of certain variables, statements, procedures, or other data can be analysed. This is very useful as a lightweight check for massive maintenance transformations.

The FC check can act as a sort of sanity check, but can be fine-tuned to gain more confidence in massive changes. In a transformation project, the input and output program can be compared on the basis of characteristics of these frequencies. For instance, if a transformation changes the value of specific variables, there should not be any changes to other variables, and the frequency of variables, statements and procedures should remain the same after the transformation. If the value is modified using an assignment statement, the frequency of that statement should increase, whereas the frequency of other statements should remain unchanged. Another example where we can use the FC check is in the Restructuring project. If a goto-elimination or dead code removal transformation is applied, code analyses can be used to check suspicious removal or duplication of certain statements.

The above examples may appear to be trivial at first sight, but the FC check can be used to check heuristics and other sanity criteria, which can detect errors at an early stage and increases the confidence in a massive transformation at low cost. Our approach to perform the FC check consists of several steps. We start by predicting possible characteristics for the frequency of certain elements in the program code after a transformation. One can try to derive them automatically from the transformation rules, but this would require a thorough analysis of the rules, i.e. in fact simulating the transformation system, which we do not consider to be a lightweight approach. Also, this approach will miss possible errors in other phases of the transformation process, and any errors in the transformation rules will be propagated to the predicted frequency characteristics. Instead, we determine the characteristics from the requirements of the transformations, such as particular changes to specific variables. Then, we analyse the program code of a system before and after transformation. If the analysis reveals unexpected changes in the frequencies, we go one step further, to the level of a single program. Subsequently, we try to identify the variables, statements or procedures that may indicate a potential error.

Prediction of characteristics Our first step is to predict the frequency characteristics for a particular transformation. Depending on what will be transformed, we can predict which frequencies will be affected, and what the characteristics for these frequencies should be. Table 6.2 shows some examples of predicted characteristics for several transformations. We briefly explain the table.

Table 6.2: Examples of predicted Frequency Characteristics ($_0$ = before, $_1$ = after).

Transformation type	Statements	Variables	Procedures
- dead code removal	$all_0 \geq all_1$ $all_1 \geq all_0 * 0.9$	$total_0 \geq total_1$	$total_0 \geq total_1$
- goto-elimination	$goto_0 > goto_1$ $other_0 \leq other_1$ $other_1 \leq other_0 * 1.1$	$total_0 = total_1$	$total_0 = total_1$
- procedure extraction	$calls_0 \leq calls_1$	$total_0 = total_1$	$total_0 = total_1$
- change variable value	$all_0 = all_1$	$total_0 = total_1$	$total_0 = total_1$
- replace variable X by variable Y	$all_0 = all_1$	$total_0 = total_1$ $varX_1 = 0$ $varY_1 = varY_0 + varX_0$	$total_0 = total_1$
- add statement X	$statX_0 < statX_1$	$total_0 = total_1$	$total_0 = total_1$

The characteristics are denoted by the frequency before transformation (e.g. $total_0$) and the frequency after transformation (e.g. $total_1$). A transformation that removes dead code results in frequencies that can only decrease or remain unchanged. We can predict an additional characteristic for a maximum decrease of statements of 10 percent ($all_1 \geq all_0 * 0.9$) when we expect that no more than 10 percent of the statements is dead code. This is just a heuristic that can be adjusted if too many false positives are found. Elimination of goto statements implies that the frequency of goto statements will decrease, and frequencies of other statements may increase due to code duplication. An additional characteristic therefore expects a maximum increase of 10 percent of the frequency of statements ($other_1 \leq other_0 * 1.1$). A transformation that extracts procedures will replace procedures by procedure calls, thus increasing the frequency of procedure calls. Changing a variable value implies that all frequencies remain unchanged, and replacing a variable means that the new frequency of the original variable will be zero ($varX_1 = 0$) and the new frequency of the replacing variable will increase by the initial frequency of the original variable ($varY_1 = varY_0 + varX_0$). A transformation that adds a certain statement implies an increase in its frequency, and no changes to other frequencies. For some of these transformations, the predicted characteristics may be obvious. But the characteristics provide insights and confidence into a large amount of transformed program code, and can detect transformation errors.

Identifying potential errors After predicting the frequency characteristics, a code analysis tool analyses the program code of the entire system before and after a transformation. If frequencies deviate from our predictions, we identify the programs that contain the unexpected changes. Then, after the programs have been identified, it may still take some effort to tell where the possible error occurred. If the program is about 10 thousand lines of code, and if the frequency of certain statements has changed from 500 occurrences to 501, it is not always easy to find this extra occurrence. This is especially true if the transformed code is very different from the original code (e.g. in a restructuring or language conversion project). For this step of identification, lexical Unix utilities can be very helpful because automated changes are carried through in a consistent way; hence, the number

of syntactic possibilities is small. In a mass modification project where small changes are made locally, an extra occurrence of a certain statement can be identified using `diff`. Checking whether a statement or variable has been modified can be done using `grep`. A combination of these Unix utilities can be used to check more complex transformations.

Summary The FC check can be a very effective way to control transformations that have been applied to a large amount of code. Although the check requires manual work, such as predicting the characteristics, it can be applied quickly and at low cost. In Section 6.5, we will demonstrate how we spotted potential errors that were revealed by irregular frequency characteristics.

6.4.2 Compilation & Regression Tests

If all else succeeds then a final check is to compile the transformed program on a suitable compiler and run regression tests. As we argued in the introduction, this check will usually be performed at a customer's site and is therefore more expensive to carry out. Hence, the CRT check is performed after other checks succeeded, and can be used to detect errors that were not detected earlier. Compilation of the transformed code can detect a wide spectrum of errors, in particular syntactic and semantic ones. For instance, in Cobol there is a restriction on the length of identifiers. If a transformation transforms an identifier to more than 30 characters, the compiler will report an error. Also, in some versions of Cobol, statements must be located in specific columns. A transformation or postprocessor may format code differently, violating this restriction. The compiler can report such problems. Duplicated procedure-names and undeclared or redundant variables can also be detected by a compiler, as well as other errors. This kind of errors can be found by compiling the transformed program. After a successful compile, running regression tests is obviously also useful for finding errors.

6.5 Case study

We applied our checks in the mass maintenance projects from chapters 4 and 5. In the Btrieve project, we applied all checks except the CFI check, since the control-flow of updated programs was changed by the transformations of that project and thus not kept invariant. However, the check could be applied to verify that the control-flow of updated programs was indeed *not* kept invariant. In the Restructuring project, we applied all of our checks. To perform the checks, we implemented them in several tools. More details on these tools and the application are given in Section 6.6.

6.5.1 The Btrieve project

In the Btrieve project, we applied the VC, GTG, FC and CRT check. The transformations consisted of a set of general-purpose transformations, including conversion and comparison functions, and a set of project-specific transformations. Here, we consider the transformations rules for the analysis and declaration of variables, and the rules that made

changes to the database calls. In total, this comprised 110 transformation rules. The transformations that performed actual changes to Cobol code were made up of only a few rules; these rules used the other rules to perform comparisons, conversions and verify preconditions. We already touched upon checks for the Btrieve project in Section 4.3.4 and we will now discuss the checks in more detail.

VC

Out of the 110 transformation rules, 75 failed the VC check. The reported rules turned our to be analysis functions. It was quite natural that these were reported, as an analysis function usually returns an argument of a different type than its input argument (e.g. to count gotos, the input is a program with goto statements and the output is an integer). Hence, analysis functions are usually erasing rules. If an input argument is not used in the condition, the rule and input argument are reported by the check. In this project, there were many analysis functions; therefore, a great deal of false positives was reported. An idea would be to distinguish between analysis functions, or to explicitly record for each erasing rule which variables are erased. Nevertheless, the VC check was quickly implemented and applied, and by looking at the reported variables we gained more confidence in the correctness of the analysis functions.

GTG

For 110 transformation rules, 172 cases were generated. There were 17 non-matching cases and 155 matching cases. On average, there were less than two cases generated per rule, which was due to the relative small patterns in the rules (i.e. the low number of variables). As we will see in the next section, the transformation patterns from the Restructuring project were larger (i.e. contained more variables), which results in more cases for each rule.

Not all generated cases were useful, as some rules, which only differed in their conditions, yielded the same cases. There were only a few of such cases, and these can (manually) be improved to cover the rules properly. Furthermore, some cases were generated as non-matching because a certain rule did not match, but such cases can be matched by a variant of the same transformation. This behaviour can be detected automatically by applying the transformation to all non-matching cases, and if cases are affected by the transformation then these were matching cases.

When we examined the non-matching cases for `Find-value`, which finds the initial value of a variable, we found that the transformation rule did not cover all possible cases that can occur according to the used grammar. For instance, the condition-value-clause in Cobol, which can be used for flag variables, was not analysed properly. For the Btrieve project, this was not a problem, as we did not have to analyse such variables, but if the function is reused it can cause an error. A more serious error that could have occurred in the Btrieve project was detected by a non-matching case for the data-value-clause. The format of this clause, taken from the VS Cobol II grammar, is shown in Figure 6.18. The optional keyword `IS` was not covered in the transformation rule; hence, Cobol variables of this format were not correctly analysed. In addition to this problem, we also found a transformation without generated cases. It turned out that this transformation was declared

```
"VALUE" [ "IS" ] Literal -> Data-value-clause


Find-value( VALUE Literal ) = Literal
```

Figure 6.18: Data-value-clause production rule from the VS Cobol II grammar and the transformation rule for finding the initial value of a variable. The `IS` keyword is optional in the production rule but not taken into account in the transformation rule.

but no transformation rules were specified; hence, no cases were generated. We removed the declaration, as it was not used.

The GTG check showed to be useful for finding deficiencies in transformations of this project, and we were able to generate unittests for documenting the behaviour of transformation rules automatically. We saved time because we did not have to construct cases manually, which is also prone to errors.

FC

We applied a FC check by comparing the code before and after the upgrade transformation. Since in this project small changes were made locally, several analyses can be done with simple Unix utilities such as `diff` (difference between 2 files) and `grep`. We predicted that the frequency of the number of variables should be the only frequency that is subject to change. In addition, the only statements that were subject to change were the call statements that called the database. So our approach for checking frequencies in this project was twofold: we monitored all relevant call statements (with the proper database operations) and we extracted all differences between the original and transformed program, thereby removing all *expected* changes.

In the transformed program, we retrieved all relevant call statements that did not have the correct variable. One call statement was reported, and this call statement was placed in comments so it was not modified for that reason. The example below illustrates this, where we use `grep` to retrieve lines with specific database operations (the operations are `b-res` and `b-unl`), and then remove lines which have the variables that must be the last argument of the call (`key-0` and `acc-0` are the expected changes). The `-A` option makes sure that the subsequent line is also retrieved, since that line contains the variable we are looking for. The `-e` option specifies the pattern(s) to search for, `-v` inverts the match, thus lines not matching the pattern are reported.

```
$ grep -A 1 -e 'call BT "__BTRV" using b-res,'
            -e 'call BT "__BTRV" using b-unl,' *.CBL |
  grep -v -e 'call BT' -e 'key-0' -e 'acc-0'
TUC002.CBL:*  tet202-record, dbl-tet202, tet202-position, acc-2.
```

Hence, the only unmodified case is a comment line and thus the result of this check was promising. This simple query can be applied to 4 million lines of code very quickly, which

was required for the tight time schedule of the project, and we increased the confidence in the changes that were made by our tools. Errors that we cannot detect with this query are call statements that have the replaced variable on the third line, or on the second line but in the wrong position. With a little more effort, these cases can also be detected.

Next we retrieved all differences between the original and transformed programs, and we removed all strings that contained any of the variables that had to be modified; the result was empty, so we knew that there were no changes besides changes which involved those variables. In addition, we checked if a new variable was not declared twice by retrieving changes in the data declarations and then checking whether the new variable was not already declared in the program before transformation. This can also be detected by most compilers but we aim at detecting such errors before compiling the code to avoid costs. Two things that we did not check with the FC check, simply because it was too time-consuming as a lightweight FC check, are the following: variables that are already declared in an include file and therefore declared twice, and variables that were replaced in the incorrect place, in a call statement or any other statement. The first issue can be detected by a compiler, the second one can be detected by one of the other checks. Summarising, the FC check allowed us to gain more confidence in the transformations of this project very quickly with a combination of relative simple tools.

CRT

As soon as we were confident about the results of our transformations, we sent the transformed code to the customer. Since we did not have the appropriate compiler, we let them compile the code to detect possible errors that were not detected earlier.

In one of the transformations, we had to determine the length of a variable. In Cobol there are several datatypes which store variables in different formats. We had to take this into account when we calculated lengths. However, the compiler reported an error for a variable and this was due to an incorrect calculation of the length. It then turned out that the assumed storage space for a specific datatype was incorrect. An error was found by compiling the code but the error was not due to the transformation but to a wrong assumption.

A real error in the transformation process was also found by the compiler. The original code came from a Windows environment, and we worked in a Unix environment. The carriage return character was removed but we did not add it before we returned it to the customer. This error was then reported by the compiler, and required another release cycle of transformed sources. Our FC check did not find the error because the comparison of the input and output program was done on input code from which the carriage return character had been removed. After another release, the code was subject to tests at the customer, and no more errors were reported.

The CRT check revealed errors and shortcomings in the transformation. The check was applied by our customer since we did not have the proper resources to compile and test the transformed code.

6.5.2 The Restructuring project

In the Restructuring project, we applied the CFI, VC, GTG, CF and CRT check. The entire transformation project consisted of 25 individual transformations in three phases. Some of them were applied several times. In total, there were 230 transformation rules. We applied the program code checks on the source base of 80,000 lines of code that was at hand during the Restructuring project.

CFI

One of the transformations eliminates goto statements, and we will apply the CFI check to its rules. This transformation consists of 16 rules that eliminate various goto constructs, while keeping the control-flow of the input program invariant. We translated all input and output patterns into LTSes using our translation tool and fed this to the bisimulation checker Aldebaran.

All LTSes were branching bisimilar, except one. This rule transforms an implicit loop, using a goto statement, into an explicit loop, using a perform statement. The rule is shown in Figure 6.19, and in Figure 6.20 the corresponding control-flow graphs of the input and the output pattern are displayed. Corresponding states of the graphs are connected by dashed lines. At first we thought this transformation rule was sound. However, applying the CFI check revealed an error. In Figure 6.20 we can see that states P1, P4 and Q1 have no corresponding state. The error in the rule is caused by `Statements3` on the left-hand side, which precede the second if statement. In the input pattern, these statements are executed before `Condition2` is evaluated, whereas in the output pattern the statements are executed *after* `Condition2` is evaluated. We corrected this error as follows: we disallowed statements in that place and then the graphs were bisimilar. See the corrected transformation in Figure 6.21. The corresponding control-flow graphs are illustrated in Figure 6.22. For completeness, we give the LTS for the incorrect rule in Figure 6.23 and the LTS for the corrected rule in Figure 6.24.

After we applied the CFI check, we were more confident about the correctness of the goto-elimination rules. One of the rules turned out to contain an error, which we corrected. Next we will apply the VC check on all transformation rules.

VC

We checked if our transformation rules complied with the VC check, stating that variables appearing on only one side should be contained in a condition. Our tool for checking this reported 31 of the 230 rules together with the suspicious variables. The reported cases were explained as follows: in 16 cases variables were removed or replaced (e.g. removal of deadcode); the rest of the cases were rules for analysis functions (e.g. verifying whether a certain label is referenced or not). Analysis functions usually return an argument of a different type than its input argument (which makes them erasing by nature), so it was quite natural that these rules were reported. Again, an idea would be to explicitly record for erasing rules which variables are erased. Nevertheless, with these results, we had to inspect only 16 rules by hand for possible errors, which is 7 percent of the total number

```
⟹
eliminate-goto(
 Label.                          ⟶  Label.
  Statements1                        Statements1
  IF Condition1                      PERFORM TEST BEFORE UNTIL
   Statements2                       Condition1 OR NOT Condition2
  ELSE                                 Statements3
   Statements3                         Statements4
   IF Condition2                       Statements1
    Statements4                      END-PERFORM
    GO TO Label                      IF Condition1
   ELSE                                Statements2
    Statements5                      ELSE
   END-IF                              Statements5
   Statements6                         Statements6
  END-IF                             END-IF
  Statements7.                       Statements7.
)
```

Figure 6.19: A transformation rule which modifies the control-flow. On the left-hand side, `Statements3` is always executed before `Condition2` is evaluated. On the right-hand side, `Condition2` is evaluated before `Statements3` is executed.

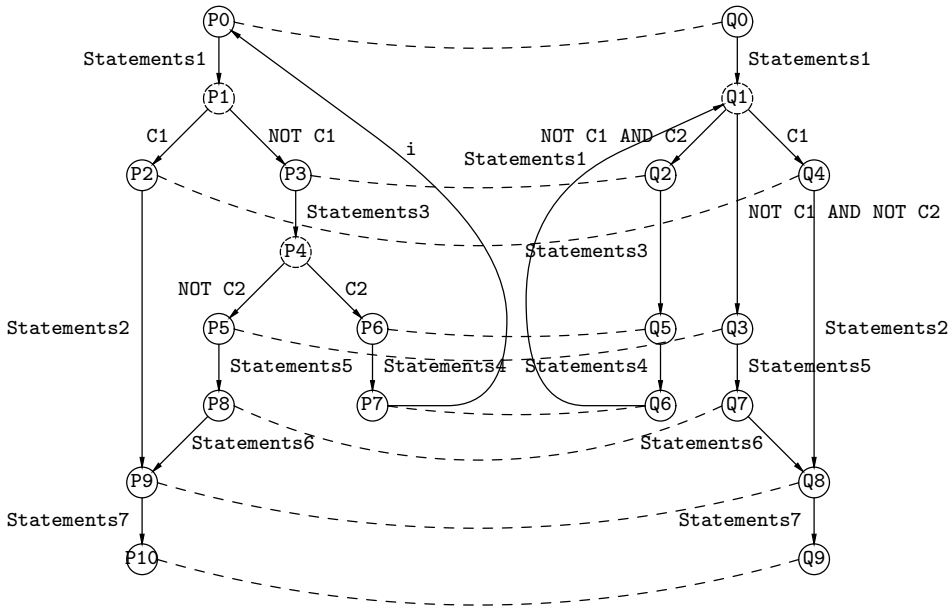

Figure 6.20: Control-flow graphs of the transformation rule from Figure 6.19; `P1`, `P4` and `Q1` have no corresponding states.

```
⟹
eliminate-goto(
 Label.                          ⟶  Label.
   Statements1                        Statements1
   IF Condition1                      PERFORM TEST BEFORE UNTIL
    Statements2                       Condition1 OR NOT Condition2
   ELSE                                 Statements3
     IF Condition2                      Statements1
       Statements3                    END-PERFORM
       GO TO Label                    IF Condition1
     ELSE                               Statements2
       Statements4                    ELSE
     END-IF                             Statements4
     Statements5                        Statements5
   END-IF                             END-IF
   Statements6.                       Statements6.
)
```

Figure 6.21: Corrected version of the transformation rule from Figure 6.19. No statements are allowed between the first `ELSE` keyword and the second if statement.

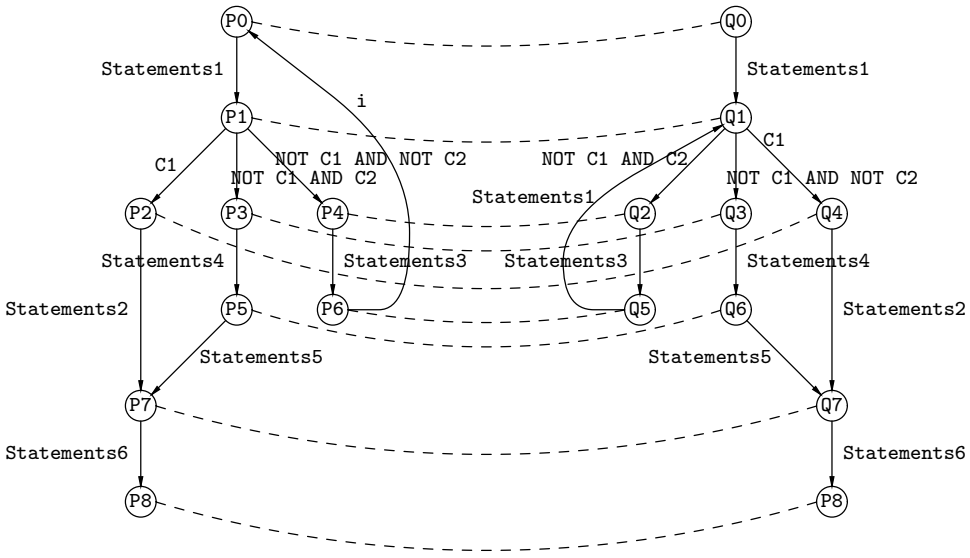

Figure 6.22: Control-flow graph of the corrected transformation rule from Figure 6.21.

Input	**Output**
des (0, 12, 11)	des (0, 10, 10)
(0, Statements1, 1)	(0, Statements1, 1)
(1, C1, 2)	(1, C1, 4)
(1, NOT C1, 3)	(1, NOT C1 AND NOT C2, 3)
(2, Statements2, 9)	(1, NOT C1 AND C2, 2)
(3, Statements3, 4)	(2, Statements3, 5)
(4, C2, 6)	(5, Statements4, 6)
(4, NOT C2, 5)	(6, Statements1, 1)
(6, Statements4, 7)	(3, Statements5, 7)
(7, i, 0)	(7, Statements6, 8)
(5, Statements5, 8)	(4, Statements2, 8)
(8, Statements6, 9)	(8, Statements7, 9)
(9, Statements7, 10)	

Figure 6.23: LTSes for the transformation rule from Figure 6.19.

Input	Output
des (0, 10, 9)	des (0, 10, 9)
(0, Statements1, 1)	(0, Statements1, 1)
(1, C1, 2)	(1, C1, 4)
(1, NOT C1 AND C2, 3)	(1, NOT C1 AND C2, 2)
(1, NOT C1 AND NOT C2, 4)	(1, NOT C1 AND NOT C2, 3)
(2, Statements2, 7)	(4, Statements2, 7)
(3, Statements4, 5)	(3, Statements4, 6)
(4, Statements3, 6)	(2, Statements3, 5)
(6, i, 0)	(5, Statements1, 1)
(5, Statements5, 7)	(6, Statements5, 7)
(7, Statements6, 8)	(7, Statements6, 8)

Figure 6.24: LTSes for the transformation rule from Figure 6.21.

```
occurs-in( Label, Reachable-labels) == false
⟹
eliminate-dead-code(
  Paragraphs1                    ⟶  Paragraphs1
  Label.                             Paragraphs2
    Statements.
  Paragraphs2
, Reachable-labels
)
```

Figure 6.25: Transformation rule which removes a variable.

```
⟹
unfold-label(
  PERFORM Label1                 ⟶  PERFORM Label2
, Label1
, Label2
)
```

Figure 6.26: Transformation rule which replaces a variable.

of rules. Hence, the VC check quickly narrowed our search space for erroneous rules. We found no problems in those rules, and we briefly discuss our findings here.

A manual inspection of the 16 reported rules revealed that in none of the cases were variables removed or replaced unintentionally. A reported case where a variable was removed is the rule in Figure 6.25, which removes an unreachable paragraph. The rule, which originates from the Eliminate-dead-code transformation, searches for paragraphs (Cobol procedures) that can never be executed, and was reported because the variable `Statements` appears on the left-hand side only. This transformation rule has no error; `Label` will be checked for being unreachable and thus the variable `Statements` can also be removed.

A reported case where a variable is replaced is shown in Figure 6.26. The rule is part of an unfolding transformation, which replaces indirect paragraph calls by direct paragraph calls. Paragraphs whose only function is to call another paragraph are removed, and all references to the removed paragraph are replaced by a reference to the called paragraph. The reported rule replaces all references to the removed paragraph (`PERFORM Label1`) by a reference to the called paragraph (`PERFORM Label2`). For this rule, `Label1` was reported because it appears only on the left-hand side. There is no error, as it is the intention of the transformation to replace `Label1` by `Label2`.

We did not find errors using our VC check. However, applying the check to 230 transformation rules resulted in only 16 rules that were suspicious and had to be inspected by hand. After we inspected the reported rules, we gained more confidence in all of the transformations, as a rule either passed the VC check or was inspected by hand.

GTG

There were 25 individual transformations, with 230 rules in total. We generated 811 cases in total. In Table 6.3, the statistics for the GTG check are shown. Note that a transformation itself consists of several rules. The relative high number of cases, compared to the Btrieve project, was due to the larger number of variables in the transformation patterns. We generated testcases for all rules. This resulted in a total of 811 cases: 432 non-matching cases and 379 matching cases. The minimum number of cases for a transformation was 3, the maximum number of cases was 258. The minimum number of matching cases was 1, the maximum number was 99. The minimum number of non-matching cases was 0, the maximum was 159. Similar to the Btrieve project, not all cases were useful since a small number of rules had the same pattern but differed in their conditions.

Most cases were generated for the goto-elimination transformation (258 cases), which had 16 individual rules. This was on average 6 matching and 10 non-matching cases per individual rule. Some of the other transformations had no non-matching cases because these covered the used grammar productions entirely. Especially the transformations that were used to normalise the code had a low number of non-matching cases, since these had to match most of the possible language constructs (e.g. the Remove-optional-keywords transformation). These transformations removed all kind of optional syntax to simplify other transformations. This resulted in many non-matching cases for the other transformations, since these assumed that the code was normalised in an earlier phase. We can improve the generation process by leaving out the normalised constructs or by automatically re-classifying them differently, or somehow encode these dependencies in the rules.

The GTG check does not take conditions of transformation rules into account so some of the matching cases were reclassified as non-matching cases. These cases were simply filtered by applying the specific transformation and then checking whether the transformation matched or not, which was completely automated. Moreover, as we also observed in the Btrieve project, if a rule does not cover a certain case while another one does, it is still a matching case; in this project we did not encounter this behaviour but it should be considered when generating testcases. Summarising, the GTG check is an interesting way to generate useful testcases to test and document transformation rules quickly, and was successfully applied to the restructuring transformations.

FC

The restructuring transformation consisted of several transformations that are applied in three phases. We could measure the frequencies in the source base before and after the entire transformation process. Instead, we increased the chance of catching errors by measuring after each phase. The error detection can be even more fine-grained if we measure after each individual transformation, but that would require us to intertwine the frequency analyses with the transformations which we did not want to do at that time. A disadvantage of the coarse-grained approach is that rules can cancel each other out, for instance, one rule adds a statement while another one removes a statement of the same type. Also, the analyses can be performed on the level of a single program, instead of on a collection of programs. This is another refinement than can be implemented. In our case study, we perform the FC check on system level and after each phase of the restructuring

Table 6.3: Statistics of the generated cases in the Restructuring project.

Transformations	25
Individual rules	230
Total generated cases	811
Matching cases	379
Non-matching cases	432
Minimum number of cases for a transformation	3
Maximum number of cases for a transformation	258
Minimum number of matching cases for a transformation	1
Maximum number of matching cases for a transformation	99
Minimum number of non-matching cases for a transformation	0
Maximum number of non-matching cases for a transformation	159

transformations, and if necessary we move on to the level of a single program.

Predicting frequencies We predicted characteristics for the following frequencies: number of sections, paragraphs, (several types of) statements, extracted paragraphs and extracted statements (statements that are untangled and placed in subroutines). For the statements, we defined characteristics for the control statements if, goto and perform. We also defined characteristics for some other statements such as move, add, display and read because these are some of the most common Cobol statements.

The predictions for each phase in the transformation are given in Table 6.4. The frequencies are explained as follows: F_0 is the frequency in the original code, F_1 is the frequency after the preprocessing phase, F_2 is the frequency after the mainprocessing phase, and F_3 is the frequency after the postprocessing phase. We will make a few remarks about the characteristics.

For each section, exactly one extra section for storing extracted statements will be created during the preprocessing phase. The number of sections therefore should double ($F_0 * 2 = F_1$). During the postprocessing phase, some of these created sections will be removed, but the frequency of sections after the postprocessing phase can never be lower than the original frequency ($F_2 \geq F_3 \geq F_0$). The frequency of paragraphs can only decrease or remain unchanged because the transformations do not introduce new paragraphs ($F_0 = F_1 \geq F_2 \geq F_3$). (A later version of the restructuring transformations introduced a first paragraph for each section to improve the extraction of paragraphs, but the frequency characteristics in the tabel are based on an earlier version.) For statements: first, during the preprocessing phase they can only decrease due to dead code removal; second, during the mainprocessing phase they can increase or decrease due to code duplication and dead code removal; finally, during the postprocessing phase they can decrease. An exception is the frequency of goto statements, which may increase during the preprocessing phase (due to the elimination of go depending statements). The frequency of extracted paragraphs and statements can increase during the mainprocessing phase, and decrease during the postprocessing phase. If this frequency increases, the frequency of out-of-line perform statements will also increase (unless this frequency decreases due to dead code removal). We refer to the Restructuring project for precise details on added or removed statements.

Table 6.4: Predicted Frequency Characteristics for the three phases (F_1, F_2, F_3) of the restructuring transformations (F_0 = frequency in the original code).

Element	**Frequency Characteristics**
Sections	$F_0 * 2 = F_1 = F_2 \geq F_3 \geq F_0$
Paragraphs	$F_0 = F_1 \geq F_2 \geq F_3$
Statements	$F_2 \geq F_3$
Extracted paragraphs	$F_0 = 0, F_1 < F_2 \geq F_3$
Extracted statements	$F_0 = 0, F_1 < F_2 \geq F_3$
if	$F_0 \geq F_1, F_2 \geq F_3$
goto	$F_1 < F_2 \geq F_3$
out-of-line perform	$F_0 \geq F_1, F_2 \geq F_3$
in-line perform	$F_0 \geq F_1, F_2 \geq F_3$
add	$F_0 \geq F_1, F_2 \geq F_3$
display	$F_0 \geq F_1, F_2 \geq F_3$
move	$F_0 \geq F_1, F_2 \geq F_3$
read	$F_0 \geq F_1, F_2 \geq F_3$
write	$F_0 \geq F_1, F_2 \geq F_3$

Comparing frequencies We developed an analysis tool to analyse the code before and after each phase, and we compared the results with our predictions. All results of the code analysis are presented in Table 6.5. The numbers in the table differ slightly from the statistics we presented in Table 5.2 in Chapter 5 because they were based on a newer version of the restructuring transformations.

The frequencies for the statements were all in accordance with the predicted characteristics. We found two other frequencies that were not in accordance with the predicted frequencies (denoted in bold face in Table 6.5), one for sections (F_1) and one for paragraphs (F_1). The frequency of sections after the preprocessing phase should have doubled, but instead it increased even more. The frequency of paragraphs should have remained unchanged after preprocessing phase, but it increased. These inconsistencies are potential errors which we had to inspect. Below we will explain how we quickly identified code that is responsible for irregular frequency changes, and how we investigated the unexpected changes in the frequencies of the paragraphs and sections.

Tracking down potential errors A potential error can be tracked down quickly using a combination of Unix utilities. We did not find any suspicious frequencies for statements, but we will discuss an example where we identify the decrease of a statement frequency. Imagine that we would like to check why the move statements are reduced from $F_0 = 14,553$ to $F_1 = 14,547$ during the preprocessing phase (see Table 6.5). First, we identify the programs which have a decrease in move statements by searching in the intermediate analysis results. It turns out that in two programs the number of move statements was reduced by three. In one program of approximately 700 lines of code, it was reduced from 118 to 115 statements, and in another one of approximately 2,000 lines of code it was reduced from 495 to 492 statements. We now have to identify the statements that were removed, and we illustrate this for the largest program. We cannot simply use `diff`

Table 6.5: Results of the code analysis (irregular frequencies are in bold face).

	Original code (F_0)	**After pre-processing** (F_1)	**After main-processing** (F_2)	**After post-processing** (F_3)
Sections	876	**1,927**	1,927	1,355
Paragraphs	3,932	**4,019**	3,196	3,151
Statements	32,742	32,740	33,015	32,739
Extracted paragraphs	0	0	1,756	1,711
Extracted statements	0	0	15,283	15,120
if	4,836	4,752	4,171	4,090
goto	2,938	2,944	665	665
out-of-line perform	2,191	2,191	4,595	4,554
in-line perform	0	0	436	435
add	2,099	2,098	2,226	2,221
display	1,257	1,257	1,257	1,257
move	14,553	14,547	14,613	14,598
read	158	158	181	181
write	1,191	1,190	1,187	1,186

to compare the input and output program because a great deal of code has been changed and moved around. Instead, we start by retrieving all move statements:

```
$ grep "MOVE" original_program.CBL > move_original
$ grep "MOVE" preprocessed_program.CBL > move_preproc
```

Now we have all move statements from the code before and after the preprocessing phase. Since statements are shuffled around by the transformations, we sort them using `sort` (`-b` ignores leading blanks, which can occur due to different indentation):

```
$ sort -b move_original > s_move_original
$ sort -b move_preproc > s_move_preproc
```

Then the next step is to compare the sorted statements using `diff`:

```
$ diff -b s_move_original s_move_preproc
176a177
<   MOVE AMOUNT-PRICE IN J-TRANS-IN TO INT IN INT-OUT-LINE1
301a303
<   MOVE DESCRIPT IN TAB-FC72(SUB-FC72) TO DESCRIPT IN INT-OUT-LINE1
342a345
<   MOVE INT-OUT-LINE1 TO INT-ACC-LINE
```

These are the three removed move statements, and now we can inspect the original program using a text editor. A fragment from the original code is shown, containing two outer if statements and one nested if statement:

```
IF CODE-COMPUTE-INT = ZERO
   MOVE DESCRIPT IN TAB-FC72(SUB-FC72) TO DESCRIPT IN TRANSLINE
   IF J-COST-COMP IN J-TRANS-IN = 3 OR 4
      MOVE BALANCE-TRANS IN J-TRANS-IN TO BALANCE-TRANS IN TRANSLINE
   ELSE
      MOVE AMOUNT-DSD TO AMOUNT IN TRANSLINE.
      GO TO 0832.

IF CODE-COMPUTE-INT NOT = ZERO
   MOVE AMOUNT-PRICE IN J-TRANS-IN TO INT IN INT-OUT-LINE1
   MOVE DESCRIPT IN TAB-FC72(SUB-FC72) TO DESCRIPT IN INT-OUT-LINE1
   MOVE INT-OUT-LINE1 TO INT-ACC-LINE
   GO TO 0832.
```

The reported move statements are shown in the second outer if statement. At first sight, that if statement can be reached by normal flow of control. The indentation suggests that the first `GO TO 0832` is part of the else-branch of the nested if statement. But the separator dot after `MOVE AMOUNT-DSD TO AMOUNT IN TRANSLINE` in the nested if statement closes all previous if statements; hence, the second outer if statement is always skipped by the first `GO TO 0832` statement. Therefore, the change in the frequency of the move statements was due to the removal of dead code. Furthermore, the irregular frequency can be an indication of an error in the code. Someone might think that the goto statement was part of the nested if statement, and the dead code is not supposed to be dead at all. So, with the irregular frequency characteristics, it is also possible to detect suspicious code that may indicate an error.

We have shown how to track down a change in a frequency in 80,000 lines of code quickly and in a semi-automatic way. We will now explain the unexpected changes for paragraphs and sections were shown in Table 6.5.

The change in the frequency of paragraphs appeared to be correct, but the prediction was wrong. We predicted that the frequency of paragraphs should have remained after the postprocessing phase. However, there was one transformation which creates extra sections, and this rule also adds one paragraph to signal the end of the program. The frequency of paragraphs therefore increased by exactly 87, which is the number of programs that were transformed. The predicted characteristic was incorrect and should have been $F_0 * 2 + P = F_1$, where P is the number of programs.

The other unexpected change, the change in the number of sections, turned out to be an error. We had predicted that the number of sections would double because for each section an extra section is be created for subroutines. The frequency increased a great deal more. Inspection of the code revealed an error in a transformation. This transformation normalises the code by adding missing section labels, but it also added a label when it was not necessary, resulting in an increase in sections. The transformation used to be correct but due to a modification in the grammar it broke down. A production was changed in order to optimise the grammar. Before the change, a sort had an optional non-terminal in its production. After the change, an intermediate sort was introduced in the grammar, and the optional non-terminal was injected into this new sort, replacing the optional non-terminal in the original production. To repair the transformation, we had to add one condition to a transformation rule that states explicitly that a particular variable is

```
PROCEDURE DIVISION.

0000-MAIN.
     DISPLAY '** PROGRAM11 BEGIN **'.
     ...
```

Before transformation

```
PROCEDURE DIVISION.
FIRST-SECTION-OF-PROGRAM SECTION.
0000-MAIN.
     DISPLAY '** PROGRAM11 BEGIN **'.
     ...
FIRST-SECTION-OF-PROGRAM-SUBROUTINES
                                SECTION.
```

After transformation

Figure 6.27: A label has been transformed to too many characters.

not present. This error should have been detected by a proper testcase, but at that time the testcases were constructed manually, and this particular case was not tested. We mention that the error was one of the reasons to develop the GTG check, in order to detect whether a grammar change breaks down a transformation.

CRT

The programs were written in mainframe Cobol, but we did not have the specific compiler at hand. We also did not have access to all include files, external utilities, and any input or test files. Nevertheless, we modified a few programs and were able to compile them on a Unix Cobol compiler. We had to remove the calls to external utilities and created some input and output files since we did not have access to that (confidential) data.

The compiler reported section and paragraph labels in the transformed program that were more than 30 characters; it turned out that two transformations created section and paragraph labels which can sometimes be more than 30 characters. In Figure 6.27 we show an example of a section label that is too large after transformation. A missing section label was added (`FIRST-SECTION-OF-PROGRAM`) to normalise the code, and then an extra section for subroutines was created using this label name (`FIRST-SECTION-OF-PROGRAM-SUBROUTINES`). The second label is 36 characters. After repairing this error we created some synthetic input files and ran the original and transformed program, and both programs behaved the same. In sum, we did not have access to all necessary resources to apply the entire CRT check on our source base, but we were able to identify an error by compiling a few programs.

In the Minefield project, we also discussed the transformations from the Restructuring project. In the context of a legacy portfolio modernisation project, we transformed a large legacy application of 15 thousand lines of code. There, the company sent us the code that we transformed. After we sent them the transformed code, they compiled and tested it with success.

6.6 Cost-benefit analysis

In this chapter, we developed several tools to perform the checks we described. All tools were developed using the ASF+SDF Meta-Environment. We will now discuss how much

Table 6.6: Time to implement tools and apply the checks.

Check	Tool	Implementation	Application
CFI	`pattern-to-lts`	3 days	$<$ 1 hour
VC	`consistency-check`	2 hours	$<$ 1 hour
GTG	`testcase-generator`	3 days	$<$ 1 hour
FC	`frequency-analysis`	1 day	$<$ 1 hour

time it took to implement the tools and apply them in the cases we discussed in Section 6.5. The data on the development and application time of all tools are summarised in Table 6.6.

The first tool we developed was called `pattern-to-lts`, and it was used for translating a transformation pattern to a labelled transition system (as part of the CFI check). We applied it to 16 control-flow preserving transformation rules. The development of `pattern-to-lts` took a few days. Application to the rules involved extraction of the left-hand side and right-hand side (input and output pattern), translating them, and feeding the resulting LTSes to the bisimulation checker. The complete application took less than one hour.

The second tool we developed, which was called `consistency-check`, was implemented for performing the VC check. The tool takes one file with rules and reports rules with variables that do not comply with the imposed constraint from Section 6.3.2. The development was done within a few hours, and the application, was done within one hour.

The tool we developed for the GTG check was `testcase-generator`. This tool took about three days to implement, and extracts information about a transformation rule, builds matching and non-matching patterns and then generates cases. The tool was applied to the parsed transformation rules, which were already available since this is done by the ASF+SDF Meta-Environment; the entire process was completely automated. Therefore, the application of the tool took less than one hour.

The last tool we developed was called `frequency-analysis`, and it was used for analysing several types of statements, paragraphs and sections in the Restructuring project. The development of `frequency-analysis` took less than one day and the application to 80,000 lines of code was done in less than one hour.

In the case studies, we found several errors using our checks. Some people might think at this point that the effort spent on the checks does not outweigh the number of found errors. But this is not true, as is explained in the so-called paradox of cost per defect as a (programming) productivity indicator [105, p 11]: 'cost per defect is always lowest where the number of defects found is greatest, and always highest where the number of defects found is least'. According to this paradox, there are always fixed costs when detecting and removing errors from software, and the number of found errors (also) depends on the quality of the software; thus high-quality software has a higher cost per defect than low-quality software since there are more errors in low-quality software. Therefore, cost per defect cannot be used to measure the effectiveness of the detection process.

The checks we described were performed on complex transformation projects and industrial programs using tools we developed rapidly. The projects involved sophisticated

transformations that were applied to business-critical software, and for that reason we spent extra effort to detect potential errors. The costs per found defect may be high but that is not an issue, as we argued above. Although it may appear to some people that our approach detected only trivial errors, any error that is detected before transformed code is returned to a customer saves the time of another release cycle. Once the tools for carrying out the checks are developed, they can be used and reused for new and existing transformations as well. With our approach, many transformation rules and a great deal of source code can effectively be checked at relative low cost.

6.7 Conclusions

We presented a lightweight approach to check mass maintenance transformations by proposing several practical checks for detecting errors. Our approach does not prove correctness of transformations but it is an effective way to have control over mass maintenance transformations and to detect errors. We believe that our approach provides circumstantial evidence for transformation correctness.

In our approach, we proposed separate checks for transformation rules and for program code, which can be performed in a semi-automatic way. Each check aims at different types of errors in different phases of a transformation process. Five checks were proposed and implemented: CFI to detect errors in control-flow preserving transformations by automatically comparing control-flow graphs; VC to detect errors in transformation rules and its conditions; GTG to detect errors on the syntactic level by automatically generating tests based on the grammar; FC to detect errors in the entire transformation process by comparing input and output code; and CRT as a final check by compiling and testing the transformed code. Our checks were illustrated with industrial examples of maintenance transformations.

We applied our approach to two mass maintenance projects: the Btrieve project from Chapter 4 and the Restructuring project from Chapter 5. The former involved a database upgrade using 110 transformation rules and over 4 million lines of code, and the latter involved restructuring transformations for Cobol consisting of 230 transformation rules and 80,000 lines of code. We showed how we were able to quickly reduce the number of transformation rules that had to be inspected by hand, and how we detected and isolated errors in transformations and large amounts of source code. Furthermore, we automatically generated testcases for the transformation rules, which was feasible and very useful for the individual rules. Not all checks detected errors in the projects, and several false positives were reported. However, using the checks we gained more confidence in the applied transformations and we were able to control the application to large systems. In conclusion, we have developed an effective lightweight approach for checking mass maintenance transformations at low cost.

Acknowledgements Many thank are due to Chris Verhoef for his ideas, comments and overall support, and thanks are also due to Steven Klusener for his contributions, especially the idea of using bisimulation to compare control-flow graphs of transformation rules. We also would like to thank Ralf Lämmel and Ernst-Jan Verhoeven for their ideas

and fruitful discussions. We are grateful for the comments that we received from the anonymous referees of the journal "Science of Computer Programming".

Road map In the final chapter, we summarise the thesis and present concluding remarks.

Chapter 7

Summary and concluding remarks

This chapter completes the thesis. We summarise our findings and present concluding remarks.

7.1 Summary

In this thesis, we described the development and deployment of tools for industrial mass maintenance. We employed automatic analysis and modification tools in several areas in software engineering and maintenance. A brief summary of each topic is given below.

Coding standards and programming errors (Chapter 3)

We have investigated the relation between coding standard violations and programming errors. By analysing violations of a certain standard in several business-critical applications, we found that many of the reported cases were programming errors. It turned out that either these errors had not yet occurred or that they had not been properly repaired in the past. The errors, which were the result of inconsistent manual changes, can jeopardise the proper operation of these applications and hinder maintenance.

The coding standard violations that were not classified as errors concerned complex programming logic. Usually, this logic does not pose a direct threat to an application, but is prone to errors and can complicate modifications.

The research showed that a relation can exist between a coding standard violation and the presence of programming errors and error-prone code. Furthermore, the research showed that automatic tools can detect this kind of problems prematurely, and they can be isolated using restructuring techniques, such as the technique we presented in Chapter 5.

Mass maintenance of a software portfolio (Chapter 4)

We have investigated how automatic tools can be deployed to carry out large-scale systematic changes, and how such tools exercise some control over the deterioration of evolving software. We discussed the entire mass change process and implemented a factory infrastructure that processed all applications of a company's software portfolio. The project illustrated that a systematic change to an application portfolio of over 4 million lines of code can be carried out in a consistent and cost-effective way.

The research illustrated that mass change tools must have a high degree of flexibility and repeatability to implement the changing requirements in mass maintenance projects. Furthermore, it appeared that a manual approach in such projects can lead to severe problems, because it is prone to errors, inconsistent, inflexible, and inefficient.

Code restructuring to allow evolution (Chapter 5)

We have investigated what can be done when a Cobol application has evolved into a complex entity due to numerous subsequent changes. We explored how we can improve and extend an existing restructuring algorithm, such that it can be applied on an industrial scale. By extracting procedures, the algorithm transforms programs with complicated programming logic into small components with isolated functionality. The restructured logic allows existing legacy applications to be modified and to evolve more easily. This is particularly useful when an existing application must be modified or migrated and (part of) the functionality must be unlocked.

The restructuring algorithm has been applied and evaluated in two case studies with large industrial applications. The research showed that the algorithm has industrial strength and is scalable for applications of over one million lines of code.

Verification of large-scale changes (Chapter 6)

We have investigated how we can exercise cost-effective control over large-scale automated changes. In an industrial environment, it is expensive to verify large-scale changes by using regression tests, and it is not cost-effective to use a formal approach for checking such changes. This is because many legacy languages, like Cobol, do not have a single semantics, which complicates the development of formal proofs. To prove that a program behaves the same before and after a transformation, it is necessary to formalise its behaviour. This requires a semantics, which in many cases can only be provided by the used compiler. Furthermore, industrial projects are often neither semantics nor correctness preserving. At the other side of the verification spectrum we find compilation and regression tests for the modified application. These brute force approaches are often expensive and time-consuming processes, but usually the most practical and logical ways to control large-scale changes.

We developed a lightweight approach for checking large-scale transformations, which is in between a formal technique and a brute force strategy. With the developed approach, several checks can be performed, ranging from sanity checks to the employment of bisimulation techniques. The checks were applied and evaluated in a case study with the projects from chapters 4 and 5.

The lightweight approach cannot detect all possible errors in mass maintenance transformations, but are cost-effective and increase the confidence in mass changes on millions of lines of code.

7.2 Concluding remarks

Capabilities and necessity of automation: increased control on complexity Evolution increases the complexity of software, but automated support of massive software maintenance opens up a number of possibilities for maintenance that are not feasible or possible otherwise. The development of new software can be affected by accurate insights in existing software. Large-scale analyses that go beyond measuring the lines of code or statements can be necessary and very valuable (e.g. function points metrics to justify IT investments), but are expensive and difficult to carry out solely by hand. Although both manual and automated changes increase the complexity of software, massive changes that are carried out without proper automated support can introduce inconsistencies and concealed errors. This increases the complexity of evolving software unnecessarily. Automatic tools and techniques can make versatile analyses and reliable modifications, and are able to free existing applications from a monolithic state. When changes are applied in a factory-like consistent way, the increasing complexity of evolving software can be controlled more precisely.

Awareness of automation The awareness of software maintenance automation in industry has not explicitly been studied in this thesis. However, we hope that the practical and industrial nature of the thesis stimulates and increases awareness of automated mass change approaches.

Role of the maintenance programmer The maintenance programmer in software automation is the domain expert of a system, which has an indispensable role when it comes to mass changing the system. Programmers, as they are in fact users of the automatically changed source code, should be involved as much as possible to increase their commitment to the modified system. Furthermore, they have valuable knowledge about a system that can significantly reduce the time to build a mass change tool and to carry out the change. When large-scale structural analyses and tedious mass modifications are performed with significant automated support, the resources of a programmer can be spent on more creative and challenging tasks.

Maturity of available software Nearly all research in this thesis has been carried out with freely available software. Hence, industrial-strength transformation tools can be developed without proprietary software. Still, as our research also indicates, effort must be spent on actually obtaining a suitable tool and infrastructure. The upfront investment in tool development and training can be one of the barriers to the widespread adoption of mass change technology in industry.

Automated maintenance and the software maintenance process Another topic of interest is the integration of mass maintenance tools with an existing maintenance process. One of the questions is *when* to apply mass change efforts. For instance, when doing restructuring, one can restructure code after each modification. This can be done, but is probably not convenient to carry out and causes significant overhead. On the other hand, code can be restructured when it becomes too expensive to maintain, but that is difficult to assess and qualify. One can also propose to do small-scale automated restructuring efforts, such as refactoring. Still, an everlasting issue with restructuring and similar quality improving efforts is to justify the need for it, i.e. code that works properly should not be modified. Hence, the most natural moment for a mass maintenance effort like a restructuring is when major change in the software's environment is due. That is the moment to deploy mass maintenance tools to carry out massive changes.

Samenvatting

Flexibele gereedschappen voor grootschalig onderhoud van software systemen[1]

De snelle ontwikkelingen in de softwaretechnologie zijn voorbijgegaan aan het onderhoud, beheer en aanpassen van operationele software. Veel softwaresystemen zijn nog ontwikkeld in de jaren '70 en '80. Deze legacysystemen zijn ontoegankelijk en statisch omdat zij geschreven zijn in verouderde programmeertalen maar vooral ook met verouderde programmeerconventies. De onderhoudsprogrammeurs zijn meestal niet bij de ontwikkeling van het originele systeem betrokken geweest en hebben vaak slechts een globaal idee van de werking van het systeem. Het is dan ook niet verwonderlijk dat softwareonderhoud een kostbare aangelegenheid is. Uit diverse studies blijkt dat meer dan de helft van de kosten van software besteed wordt na oplevering; van deze kosten wordt weer de helft besteed aan het analyseren van de code, een kwart aan testen en slechts een 5% aan het doorvoeren van de eigenlijke wijziging. Softwareonderhoud kan efficiënter worden ingericht door de inzet van automatische gereedschappen. Hierbij is het essentieel dat deze gereedschappen snel kunnen worden aangepast aan de kenmerken van de onderhavige onderhoudsproblematiek, en dat de software-engineers die deze tooling ontwikkelen de onderhoudsproblematiek in detail begrijpen.

Veranderingen

Softwaresystemen zijn constant onderhevig aan veranderingen en worden daardoor steeds complexer. Deze aanpassingen blijven niet beperkt tot het uitbreiden van een datum voor het jaar 2000 of invoering van de euro. Vele (grootschalige) veranderingen zijn aan de orde van de dag. Bijvoorbeeld een nieuwe versie van een besturingssysteem, een databasemigratie, een verandering in de renteberekening, of het zoeken en verwijderen van fouten. Deze aanpassingen worden gedaan door (verschillende) onderhoudsprogrammeurs en resulteren in groeiende, steeds complexer wordende systemen. Door deze *software evolutie* vergt iedere aanpassing steeds meer tijd en brengt dus hogere kosten met zich mee.

[1] Delen van deze samenvatting zijn gepubliceerd in: "Automatiseren complex maar lucratief", Steven Klusener en Niels Veerman, Informatie, September 2003.

Onderhoudsproblematiek

De onderhoudsproblematiek, vooral die van legacysystemen, verschilt in grote mate van de software-engineering zoals onderwezen wordt op de universiteiten en hogescholen. Talen als Cobol en jcl's (job control language) zijn de standaard bij het merendeel van de grote softwaresystemen, en niet de moderne object-georiënteerde talen als Java of C++, of zelfs de meer klassieke procedurele talen als Pascal en C. Eenvoudige concepten uit het abstractiebeginsel van de software-engineering, zoals procedures en lokale variabelen (in Pascal en C) of de meer geavanceerde object-oriëntatie uit Java en C++ zijn niet beschikbaar in de Cobolvarianten Cobol'74 en Cobol'85. In deze talen zijn de legacysystemen vaak geschreven. (Deze concepten zijn overigens wel tot op zekere hoogte beschikbaar in de laatste versie van Cobol, Cobol2002, maar introductie van deze concepten in de legacysystemen zal vele complexe aanpassingen vereisen.) Hierdoor kan een analyse van een Cobol-fragment meer tijd kosten dan een fragment in een andere taal. De onderhoudsprogrammeur moet immers uitzoeken welke plaats het fragment inneemt in de control-flow van het gehele programma, en waar de variabelen die erin voorkomen nog meer worden gebruikt (alle variabelen zijn immers globaal). De data van een legacysysteem zijn veelal niet opgeslagen in relationele databases maar in hiërarchische databases als IMS of zelfs in een groot aantal platte files. Merk op dat hiërarchische databases, net als Cobol, al sinds tientallen jaren als verouderd beschouwd worden en daarom niet worden onderwezen.

Vooral de echt grote databases, van bijvoorbeeld banken en overheidsinstanties met persoonsgegevens van miljoenen mensen, zijn vaak nog hiërarchisch; zo wordt gezegd dat de meerderheid van de data wereldwijd nog worden bijgehouden in hiërarchische databases. Sourcecode (broncode) van deze complexe systemen moet worden aangepast in eenvoudige teksteditors. De onderhoudsprogrammeur heeft vaak geen 'intelligente' programmeeromgeving zoals Delphi, Visual Studio, of Visual Age tot zijn beschikking, waardoor het aanpassen van de naam van een variabele al een tijdrovende actie wordt. Verder zijn legacysystemen over het algemeen groot; zij bestaan veelal uit duizenden programma's van elk duizenden (of zelfs tienduizenden) regels code, hun totale omvang varieert tussen de honderdduizend en enkele miljoenen regels code. Hierbij kunnen nog duizenden copybooks (include files), schermdefinities, record- en tabeldefinities en batchjobs komen. Er zijn nog meer verschillen met de modernere software-engineering, zo kunnen voor documentatie, testen, versiebeheer en vele andere aspecten soortgelijke verschillen worden aangegeven. Deze verschillen leveren ook een cultuurverschil op tussen de it'ers die zich met systeemonderhoud bezighouden en de it'ers die zich met 'moderne' softwaretechnologie bezighouden. Om het onderhoudsproces effectiever in te richten is het van belang om dit cultuurverschil te overbruggen. Dit kan bereikt worden door onder meer over te gaan van handmatig naar grotendeels geautomatiseerd softwareonderhoud.

Risico's

Het analyseren en aanpassen van legacysoftwaresystemen voor onderhoud gebeurt vooralsnog grotendeels handmatig. Echter, bij deze aanpak speelt een aantal risico's. In het begin van een grootschalig aanpassingstraject gaat men bij de probleemdefinitie vaak uit van de reguliere gevallen die veranderd moeten worden. Pas bij de daadwerkelijke uitvoering komen de uitzonderingsgevallen aan het licht. Hierna wordt de probleemdefinitie

aangepast. Dit houdt echter wel in dat men telkens opnieuw moet beginnen. De probleemdefinitie kent ook vaak interpretatieverschillen zonder dat dit de betrokkenen duidelijk is. Binnen een team van onderhoudsprogrammeurs levert dit verschillende resultaten op, die pas laat in het project naar voren kunnen komen. Komen deze interpretatieverschillen eenmaal naar voren, dan moet de probleemdefinitie worden aangescherpt en kan men weer opnieuw beginnen. Verder is uit de literatuur bekend dat bij ongeveer 10% van de tekstaanpassingen een typefout wordt gemaakt, wat de kans op fouten bij een handmatige aanpassing groot maakt en inconsistenties in de software introduceert. Om de genoemde risico's enigszins op te vangen moet er een uitgebreid testproces worden ingericht, dat dikwijls tijdrovender is dan de uitvoering van de aanpassing zelf. Al met al kost een handmatige aanpak vaak niet alleen veel tijd maar is het ook nog eens foutgevoelig.

Flexibele automatische gereedschappen

Veel van de analyses en aanpassingen voor grootschalig onderhoud zijn met de juiste technologie gedeeltelijk of geheel te automatiseren met gereedschappen. Deze gereedschappen zijn computerprogramma's die een te analyseren of aan te passen programma als invoer nemen, en daaruit een rapport of een aangepast programma genereren. Vooral bij grote systemen (meer dan 10 duizend tot meer dan 1 miljoen regels sourcecode) is een automatische benadering interessant omdat het vaak om kleine veranderingen gaat, die op vele plaatsen moeten worden uitgevoerd. De voordelen van een automatische aanpak zijn vooral efficiëntie, snelheid en nauwkeurigheid. In het bijzonder kan met een automatische aanpak meer controle worden uitgeoefend op de toenemende complexiteit die op de lange termijn bij alle evoluerende software optreedt. Met name grootschalige structurele aanpassingen die met de hand uitgevoerd worden introduceren inconsistenties in de sourcecode, wat leidt tot verdere toename van de complexiteit van software. Een automatisch gereedschap biedt meer controle en in zekere mate de garantie dat een verandering op een consistente wijze doorgevoerd wordt. Voordat een automatisch gereedschap ontwikkeld wordt, moet de probleemdefinitie tot in detail duidelijk zijn. Bij eventuele nieuwe inzichten zijn de gereedschappen snel aan te passen. Bovendien is het aantal regels code dat uiteindelijk wordt ingetypt om een automatische aanpassing te specificeren aanzienlijk kleiner dan het aantal dat nodig is bij een handmatige aanpak, wat de kans op (type)fouten verlaagt.

Generieke taaltechnologie

Generieke taaltechnologie is in veel gevallen de basis om snel en flexibel gereedschappen te fabriceren voor nauwkeurige automatische analyses en aanpassingen. De ontwikkeling van automatische gereedschappen in de standaard programmeertalen neemt anders te veel tijd in beslag of is simpelweg niet mogelijk. Generieke taaltechnologie kan een aantal benodigde basisfunctionaliteiten genereren waardoor veel werk uit handen genomen wordt. Voordat een aanpassing of analyse kan worden uitgevoerd moet de sourcecode worden geparseerd (ontleed), zoals men zinnen ontleedt in een onderwerp, persoonsvorm, lijdend voorwerp et cetera. Dit gebeurt aan de hand van een grammatica die de betreffende taal beschrijft. Grammatica's vormen een essentieel onderdeel bij het analyseren en aanpassen van sourcecode. Van sommige talen is de grammatica verkrijgbaar via het internet of uit

handleidingen, maar deze vereisen aanpassingen en zijn meestal niet direct te gebruiken. Bij het opstellen of aanpassen van een grammatica kan het goed zijn om rekening te houden met de wijziging die in de sourcecode moet worden doorgevoerd. Het parseren vertaalt de sourcecode in een parse tree – een boomstructuur waarin de elementen van de code gerepresenteerd zijn in de structuur zoals ze in de code voorkomen, geclassificeerd naar hun soort. Een voorbeeld van een soort is een statement (een opdracht). Het voordeel van de parse tree is dat nu eenvoudig gezocht kan worden naar patronen met bepaalde taalconstructies, bijvoorbeeld een bepaald type statement, om deze vervolgens aan te passen.

De eigenlijke analyses en aanpassingen kunnen met behulp van generieke taaltechnologie worden uitgeschreven als transformatieregels met patronen aan de linker en rechterkant. Als de linkerkant in de (parse tree van de) sourcecode voorkomt, dan wordt deze vervangen door de rechterkant. Door in de patronen variabelen te gebruiken om taalconstructies te representeren, kan worden volstaan met een minimum aantal patronen. In elke moderne teksteditor kan men zoek-en-vervangopdrachten geven, waarbij het ene woord wordt vervangen door het andere. De patronen in de generieke taaltechnologie kunnen worden beschouwd als 'intelligente' patronen, waarbij rekening gehouden wordt met de structuur van de taal (de grammatica) en de context waarin de aanpassing en analyse moeten worden uitgevoerd. Zodra de aanpassingen zijn gespecificeerd en de tooling is ontwikkeld, kan deze op de sourcecode worden toegepast. Dit gebeurt met een infrastructuur waarin op een fabrieksmatige wijze analyses en aanpassingen kunnen worden uitgevoerd. Afhankelijk van de grootte van het systeem en de complexiteit van de aanpassing kan de toepassing variëren van minuten tot enkele uren voor een geheel systeem. Tot slot kunnen nog enkele geautomatiseerde controles uitgevoerd worden die specifiek voor deze aanpassing zijn ontwikkeld.

De ontwikkeling en de afstemming van de tooling kan enige tijd in beslag nemen. Gedurende deze periode kan het reguliere onderhoud (bijvoorbeeld het herstellen van productiefouten) gewoon doorgaan. Op het moment dat de tooling beschikbaar is wordt de meest recente versie van de sourcecode aangeleverd. Deze wordt vervolgens automatisch aangepast en binnen korte tijd opgeleverd, zodat het regulier onderhoud niet 'bevroren' hoeft te worden en eventuele versieproblematiek wordt vermeden.

Onderzoek

Dit proefschrift beschrijft onderzoek naar grootschalige automatische transformatieprojecten dat aan de Vrije Universiteit in Amsterdam wordt uitgevoerd. Er is onder meer onderzoek gedaan naar het analyseren en herstructureren van sourcecode, en het uitvoeren van systematische wijzigingen aan grote systemen op een fabrieksmatige wijze. Verder is gewerkt aan methoden om op kosteneffectieve wijze controle uit te oefenen op grootschalige automatische sourcecode transformaties. Het onderzoek is voornamelijk uitgevoerd in nauwe samenwerking met het bedrijfsleven. Aan de hand van industrieële onderhoudsprojecten is onderzocht hoe op een verantwoorde en gestructureerde manier grootschalig software onderhoud ondersteund kan worden met automatische software transformaties.

Resultaten

Het onderoek dat in dit proefschrift is beschreven is zeer toepassingsgericht. Op verscheidene gebieden binnen de software engineering en onderhoud zijn automatische gereedschappen ingezet, die hieronder besproken worden.

Hoofdstuk 3: Programmeerstandaards en programmeerfouten Er is onderzoek gedaan naar de rol van programmeerstandaards bij het vinden en voorkomen van programmeerfouten. Hiervoor is een aantal bedrijfskritische applicaties van verschillende bedrijven geanalyseerd op overtredingen van een bepaalde programmeerstandaard. Bij het analyseren van de overtredingen is gebleken dat een overtreding in veel gevallen een programmeerfout betrof. De gevonden programmeerfouten zijn mogelijk nog niet opgetreden, of in het verleden niet op een juiste manier verholpen. De fouten, die geïntroduceerd waren tijdens inconsistent handmatig onderhoud, kunnen het functioneren van een applicatie ernstig in gevaar brengen en het onderhoud bemoeilijken.

De overtredingen die niet als programmeerfouten geclassificeerd werden betroffen meestal complexe programmeerconstructies. Deze constructies vormen vaak geen directe bedreiging, maar zijn wel foutgevoelig en kunnen een aanpassing of een migratie bemoeilijken.

Het onderzoek heeft aangetoond dat er in een applicatie een relatie kan bestaan tussen overtredingen van programmeerstandaarden en de aanwezigheid van programmeerfouten en foutgevoelige programmeerconstructies. Verder is aangetoond dat automatische gereedschappen het mogelijke maken om dit soort problemen voortijdig te ontdekken en te isoleren met behulp van herstructureringstechnieken, zoals beschreven in Hoofdstuk 5.

Hoofdstuk 4: Fabrieksmatig uitvoeren van systematische aanpassingen Er is onderzocht hoe automatische gereedschappen kunnen worden ingezet om grootschalige systematische veranderingen uit te voeren, en hoe zulke gereedschappen enige invloed kunnen uitoefenen op de groeiende complexiteit van evoluerende software. Er is gekeken hoe gereedschappen op een fabrieksmatige wijze kunnen worden ingezet bij de grootschalige systematische aanpassing van een volledige portefeuille van bedrijfskritische applicaties. In het kader van een commercieel onderhoudsproject bij een financiële instelling zijn gereedschappen ontwikkeld en in een fabrieksmatige infrastructuur geplaatst, om vervolgens een database-upgrade geautomatiseerd uit te voeren. Het project heeft laten zien dat een systematische verandering in een applicatieportefeuille van meer dan 4 miljoen regels sourcecode op een consistente, veilige en kosteneffectieve wijze kan worden uitgevoerd.

Uit het onderzoek is gebleken dat automatische gereedschappen een hoge mate van flexibiliteit en herhaalbaarheid moeten hebben om voortschrijdende inzichten tijdens een grootschalig onderhoudsproject te kunnen implementeren. Verder is gebleken dat in zulke projecten een volledig handmatige aanpak tot grote problemen kan leiden, omdat die foutgevoelig, inconsistent, inflexibel en inefficiënt is. Ondersteuning door automatische gereedschappen is vaak een vereiste.

Hoofdstuk 5: Herstructureren van sourcecode om evolutie te vergemakkelijken Er is onderzocht wat er gedaan kan worden om complexe applicaties, die tientallen jaren lang

vele malen zijn aangepast, weer dus danig flexibel te maken dat verdere veranderingen eenvoudiger zijn door te voeren. Er is onderzoek gedaan naar het verbeteren en uitbreiden van en op grote schaal toepassen van een bestaande herstructureringsmethode. Met deze methode kunnen gecompliceerde programma's getransformeerd worden tot kleine componenten met functionaliteit, die op een gestructureerde wijze worden aangeroepen. Op deze manier zijn programma's eenvoudiger te begrijpen en aan te passen, en kunnen de losse componenten eenvoudiger worden hergebruikt. Dit is met name van belang op het moment dat een bestaande applicatie moet worden aangepast of gemigreerd, waarbij (delen van) de functionaliteiten moeten worden ontsloten.

De methode is toegepast en geëvalueerd in twee case studies met een aantal grote industriële systemen. Het onderzoek heeft aangetoond aan dat de methode flexibel is en geschikt is voor toepassing op industriële applicaties van enkele miljoenen regels sourcecode.

Hoofdstuk 6: Verifiëren van automatische aanpassingen Er is onderzocht hoe op kosteneffectieve wijze controle uit te oefenen op grootschalige automatische aanpassingen. Onder industriële omstandigheden is het duur om een grootschalige sourcecode transformaties te verifiëren met behulp van regressietesten, en het is niet kosteneffectief met een formele aanpak zulke aanpassingen te controleren. Dit komt ondermeer doordat voor veel legacy technologiën en programmmeertalen, zoals Cobol, geen eenduidige semantiek bestaat, wat het ontwikkelen van formele bewijzen compliceert. Bij het aantonen dat een programma voor transformatie hetzelfde gedrag vertoont als na transformatie is het nodig om het gedrag van het programma te kunnen formaliseren. Hiervoor is er een semantiek nodig, die in veel gevallen alleen door de gebruikte compiler gegeven kan worden. Verder spelen bij industriële onderhoudsprojecten semantiek en correctheid een ondergeschikte rol; het is voornamelijk van belang dat een bepaalde verandering *consistent* uitgevoerd wordt. Aan het andere eind van het verificatie spectrum staan compilatie en regressietesten van het aangepaste systeem. Deze *brute force* methoden zijn vaak dure en tijdrovende processen, maar vooralsnog de meest praktische en voor de hand liggende manieren om grootschalige aanpassingen te controleren.

In het onderzoek zijn *lightweight* technieken ontwikkeld, die het gat tussen een formele benadering en een brute force aanpak invullen. Met deze technieken kunnen foutcontroles uitgevoerd worden, variërend van *sanity checks* tot het inzetten van *bisimulatie* technieken uit de process algebra. In een case study zijn de technieken toegepast en geëvalueerd op de onderhoudsprojecten uit de hoofdstukken 4 en 5. Met de ontwikkelde technieken kunnen weliswaar niet alle mogelijke fouten in een automatische aanpak opgespoord worden, maar het slagen van de controles is een kost-effectieve middenweg en geeft meer vertrouwen in een aanpassing die op miljoenen regels code is doorgevoerd.

Conclusie

Het automatisch aanpassen van grote softwaresystemen biedt veel voordeel ten opzichte van handmatige aanpassing. Vooral voor aanpassingen die op veel plaatsen binnen een grote hoeveelheid sourcecode nodig zijn is een automatische aanpassing zeer waardevol. Een volledig handmatige aanpak is in zulke gevallen erg foutgevoelig en kost veel tijd, en

leidt tot inconsistenties in de software die op langere termijn voor problemen zorgen. Met behulp van generieke taaltechnologie kan op een gestructureerde wijze flexibele transformatie tooling ontwikkeld worden. Als eenmaal in de tooling is geïnvesteerd, zijn aanpassingen snel en nauwkeurig uit te voeren op een fabrieksmatige wijze. Het ontwikkelen van deze tooling is een technisch complexe aangelegenheid, maar biedt in veel gevallen verantwoorde en kosteneffectieve ondersteuning voor een traditioneel handmatige aanpak. Bovendien kan de toenemende complexiteit van veranderende systemen op de langere termijn op deze wijze beter in de hand worden gehouden.

Bibliography

[1] L. Abraido-Fandino. An Overview of Refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, 1987.

[2] Acucorp. AcuCobol Development System. http://www.acucorp.com.

[3] Acucorp inc. *AcuCobol-85 Reference Manual*, 1999.

[4] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[5] A.V. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] Amercian National Standards Institute (ANSI). *Information technology – Programming languages – COBOL, reference number ISO/IEC 1989:2002(E)*, 2002.

[7] E. Arranga, I. Archbell, J. Bradley, P. Coker, R. Langer, C. Townsend, and M. Weathley. In Cobol's Defense. *IEEE Software*, 17(2):70–72, 2000.

[8] E. Arranga and F.P. Coyle. Cobol: Perception and reality. *Computer*, 30(3):126–128, March 1997.

[9] D.C. Atkinson and W.G. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Software: Practice & Experience*, 36(4):413–447, 2006.

[10] AT&T and Lucent Bell Labs. Graphviz - Graph Visualisation Software, 2002. http://www.graphviz.org/.

[11] J. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume IV, pages 149–268. Oxford University Press, 1995.

[12] J. Baeten and P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[13] Project Bauhaus. Software Architecture, Software Reengineering, and Program Understanding.
http://www.iste.uni-stuttgart.de/ps/bauhaus/.

[14] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Beauty and the Beast or A Formal Description of the Control Constructs of Cobol and its Implementation. Technical Report 93.39, Department of Computer Science, University of Zurich, Switzerland, 1993.

[15] I.D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992.

[16] I.D. Baxter. DMS: Program Transformations for Practical Scalable Software Evolution. In P. Linos, M. Harman, and B. Korel, editors, *Proceedings of the 20th International Conference on Software Maintenance (ICSM'03)*, pages 625–634. IEEE Computer Society Press, 2004.

[17] I.D. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In R. Koschke, E. Burd, and P. Aiken, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 281–290. IEEE Computer Society Press, 2001.

[18] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[19] L.A. Belady and M.M. Lehman. *Progam evolution*. Academic Press Inc., 1985.

[20] K. Bennett, N. Gold, and T. Systä, editors. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*. IEEE Computer Society, 2004.

[21] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[22] Terese: M. Bezem, J.W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, NY, USA, 2003.

[23] D. Blasband. *Automatic Analysis of Ancient Languages*. PhD thesis, Free University of Brussels, 2000. Available via www.phidani.be/homes/darius/thesis.html.

[24] D. Blasband. Parsing in a hostile world. In R. Koschke, E. Burd, and P. Aiken, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 291–300. IEEE Computer Society Press, 2001.

[25] B. Boehm and V.R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, 2001.

[26] B.W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[27] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[28] J.M. Boyle, T.J. Harmer, and P.J. McParland. Applications of the TAMPR Transformation System. In S. Flynn and A. Butterfield, editors, *2nd Irish Workshop on Formal Methods*, Workshops in Computing. BCS, 1998.

[29] J.M. Boyle, T.J. Harmer, and P.J. McParland. Transformations to Restructure and Re-engineer COBOL Programs. *Automated Software Engineering*, 5(3):321–345, 1998.

[30] M. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software: Practice & Experience*, 30(3):259–291, 2000.

[31] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial Applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[32] M. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[33] M. van den Brand, P. Klint, and C. Verhoef. Core Technologies for System Renovation. In K.G. Jeffery, J. Král, and M. Bartosek, editors, *Proceedings of the 23rd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'96)*, volume 1175 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 1996.

[34] M. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[35] M. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[36] M. van den Brand, S. Klusener, L. Moonen, and J. Vinju. Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation. *Electronic Notes in Theoretical Computer Science*, 82(3):575–591, 2003.

[37] M. van den Brand, A.T. Kooiker, N. Veerman, and J.J. Vinju. An architecture for context-sensitive formatting (extended abstract). In H.M. Sneed, T. Gyimóthy, and V. Rajlich, editors, *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 631–634. IEEE Computer Society Press, 2005.

[38] M. van den Brand, A.T. Kooiker, N. Veerman, and J.J. Vinju. A language independent framework for context-sensitive formatting. In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors, *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 103–112. IEEE Computer Society Press, 2006.

[39] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.

[40] M. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE Computer Society Press, 1997.

[41] M. van den Brand, A. Sellink, and C. Verhoef. Obtaining a Cobol Grammar from Legacy Code for Reengineering Purposes. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer-Verlag, 1997.

[42] M. van den Brand, A. Sellink, and C. Verhoef. Control Flow Normalization for COBOL/CICS Legacy System. In *2nd Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 11–20, 1998.

[43] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In P. Linos, U. de Carlini, S. Tilley, and G. Visaggio, editors, *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'99)*, pages 108–117, 1998.

[44] M. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[45] M. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[46] M. van den Brand and J.J. Vinju. Rewriting with layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of RULE2000*, 2000.

[47] M. van den Brand and E. Visser. Generation of Formatters for Context-free Languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[48] M. van den Brand et al. The ASF+SDF Meta-Environment. Available at http://www.meta-environment.org.

[49] F.P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[50] D. Brown and T. Mason. *Lex & yacc*. O'Reilly & Associates, Inc., 1990.

[51] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray. *Anti-patterns: Refactoring Software, Architecture and Projects in Crisis*. John Wiley & Sons Inc, 1999.

[52] J. Brunekreef and B. Diertens. Towards a User-Controlled Software Renovation Factory. *Science of Computer Programming*, 45(2–3):175–191, 2002.

[53] CALCE: Computer-aided Life Cycle Enabling of Software Assets. http://www.cs.vu.nl/~steven/calce/.

[54] F.W. Calliss. Problems with automatic restructurers. *ACM SIGPLAN Notices*, 23(3):13–21, 1988.

[55] Case Consult. http://www.caseconsult.com.

[56] N. Chapin and A. Cimitile, editors. *Journal of Software Maintenance and Evolution: Research and Practice*. John Wiley & Sons, Ltd., 2005.

[57] E.J. Chikofsky and J.H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[58] Compaq. Compaq COBOL. http://www.compaq.com.

[59] Compaq Information Technologies Group. *Cobol Reference Manual Version 2.5*, 2002.

[60] Computer Data Systems, Inc., Superstructure/Retool, 1984.

[61] J.R. Cordy. Comprehending Reality – Practical Barriers to Industrial Adoption of Software Maintenance Automation. In H. Müller, R. Koschke, and K. Wong, editors, *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 196–205. IEEE Computer Society, 2003.

[62] J.R. Cordy, T.R. Dean, A.J. Malton, and K.A. Schneider. Source Transformation in Software Engineering using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[63] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[64] A. Cox and C. Clarke. Syntactic Approximation Using Iterative Lexical Analysis. In H. Müller, R. Koschke, and K. Wong, editors, *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 154–163. IEEE Computer Society, 2003.

[65] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for reengineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 14:257–292, 2002.

[66] R. Damaševicius and V. Štuikys. Taxonomy of the Program Transformation Processes. *Information Technology and Control*, 22(1):39–52, 2002.

[67] T. Dean, R. Koschke, and M. Van De Vanter, editors. *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*. IEEE Computer Society, 2004.

[68] A. van Deursen. The Software Evolution Paradox, Inaugural Lecture (in Dutch), February 2005, Delft University of Technology. http://homepages.cwi.nl/~arie/intreerede/.

[69] A. van Deursen. A comparison of Software Refinery and ASF+SDF. In *Program Analysis for System Renovation (Resolver Release I)*, pages 9–41. CWI, 1997.

[70] A. van Deursen, P. Klint, and C. Verhoef. Research Issues in the Renovation of Legacy Systems. In J.-P. Finance, editor, *Proceedings of the 2nd Conference on Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 1–23, Amsterdam, 1999. Springer Berlin / Heidelberg.

[71] A. van Deursen and J. Visser. Building Program Understanding Tools Using Visitor Combinators. In F. Balmas, M. Harman, and E. Merlo, editors, *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pages 137–146. IEEE Computer Society, 2002.

[72] Digital Equipment Corporation. *Cobol Reference Manual Version 2.3*, 2002.

[73] E. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11:147–148, 1968. Available at http://www.acm.org/classics/oct95/.

[74] D. Dougherty and A. Robbins. *sed & awk*. O'Reilly & Associates, Inc., 1997.

[75] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - Generic Understanding of Programs. In T. Mens, A. Schürr, and G. Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.

[76] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, and M. Tofte. AnnoDomini: from type theory to Year 2000 conversion tool. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages (POPL'99)*, ACM SIGPLAN Notices, pages 1–14, New York, NY, USA, 1999. ACM Press.

[77] ELTIS: Extending the Lifetime of Information Systems. http://www.titu.jyu.fi/eltis/.

[78] D. Faust and C. Verhoef. Software Product Line Migration and Deployment. *Software: Practice & Experience*, 33:933–955, 2003.

[79] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[80] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (Caesar/Aldebaran Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1996.

[81] J. Field and G. Ramalingam. Identifying procedural structure in Cobol programs. In W. Griswold and S. Horwitz, editors, *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE'99)*, pages 1–10. ACM Press, 1999.

[82] W. Fokkink. *Introduction to process algebra*. Springer-Verlag, Berlin, 2000.

[83] W. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories. In J.-P. Finance, editor, *Proceedings of the 2nd Conference on Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 1999.

[84] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[85] Free Software Foundation. TinyCobol. http://tiny-cobol.sourceforge.net/.

[86] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews*. Dorset House, 3rd Ed. 1990. Originally published by Little, Brown & Company, 1982.

[87] Fujitsu. NetCobol. http://www.fujitsu.com.

[88] Fujitsu software corporation. *Cobol 85 Reference Manual*, 1996.

[89] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with *dot*, 2002. http://www.graphviz.org/Documentation/dotguide.pdf.

[90] A. Garrido and R. Johnson. Challenges of refactoring C programs. In M. Aoyama, K. Inoue, and V. Raijlich, editors, *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 6–14, New York, NY, USA, 2002. ACM Press.

[91] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[92] GrammaTech. The Synthesizer Generator. http://www.grammatech.com.

[93] W.G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

[94] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT Revolution Continues: Managing Diversity in the 21st Century*. Gartner Group, 1996.

[95] M.T. Harandi. An experimental COBOL restructuring system. *Software: Practice & Experience*, 13:825–846, 1983.

[96] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[97] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notice*, 35(3):39–48, March 2000.

[98] IBM Corporation. IBM Cobol. http://www.ibm.com.

[99] IBM Corporation. *IBM Cobol for MVS & VM Language Reference*, 1995.

[100] IBM Corporation. *IBM Cobol for OS/390 & VM Language Reference, Fifth Edition*, 2000.

[101] IBM Corporation. *IBM Enterprise COBOL for z/OS Language Reference Version 3 Release 3*, 2004.

[102] IEEE: The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Software Maintenance, IEEE Std 1219-1998*, 1998.

[103] D.C. Ince. The Automatic Generation of Test Data. *The Computer Journal*, 30(1):63–69, 1987.

[104] S. Johnson. Yacc - Yet Another Compiler Compiler. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1975.

[105] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[106] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[107] C. Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.

[108] C. Jones. Software project management in the twenty-first century. http://www.spr.com/news/SoftwareProjectArticle.pdf, 1999.

[109] N. Jones. Year 2000 Market Overview. Technical report, Gartner Group, Stamford, CT, USA, 1998.

[110] M. de Jonge, E. Visser, and J. Visser. XT: A Bundle of Program Transformation Tools. In M. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[111] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice Hall, Inc., 1988.

[112] P. Klint. A Meta-Environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[113] P. Klint, R. Lämmel, and C. Verhoef. Towards an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, July 2005.

[114] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[115] S. Klusener. Source Code Based Function Point Analysis for Enhancement Projects. In S.L. Pfleeger, C. Verhoef, and H. van Vliet, editors, *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 373–376. IEEE Computer Society Press, 2003.

[116] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In S.L. Pfleeger, C. Verhoef, and H. van Vliet, editors, *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 179–188. IEEE Computer Society Press, 2003.

[117] S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.

[118] S. Klusener and C. Verhoef. 9210: the zip code of another IT-soap. *Software Quality Journal*, 12:297–309, 2004.

[119] D.E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[120] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In T. Reps and W. Wegman, editors, *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'00)*, pages 155–169. ACM Press, 2000.

[121] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos. Code Migration Through Transformations: An Experience Report. In *Proceedings of the Center for Advanced Studies Conference (CASCON'98)*, pages 1–13, 1998.

[122] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit – System Demonstration. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002.

[123] A. Krause and M. Olson. *Basics of S and S-Plus*. Springer-Verlag, 2nd edition, 2000.

[124] R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, The Netherlands, 1999.

[125] C.W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[126] R. Lämmel. Grammar Adaptation. In J.N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer-Verlag, 2001.

[127] R. Lämmel, editor. *Science of Computer Programming, Special issue on Program Transformation*, volume 52. Elsevier Science Publishers, 2004.

[128] R. Lämmel and W. Schulte. Controlled explosion in grammar-based testing. Draft, 20 pages, 24 October 2003.

[129] R. Lämmel and C. Verhoef. VS Cobol II Grammar Version 1.0.3, 1999. Available online at http://www.cs.vu.nl/grammars/vs-cobol-ii/.

[130] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[131] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software: Practice & Experience*, 31(15):1395–1438, December 2001.

[132] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 1–25. Elsevier Science, 2001.

[133] B. Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[134] LegacyJ Corporation. *PERCobol Language Reference Manual, Eight Edition*, 2005.

[135] M.M. Lehman. Laws of program evolution - rules and tools for programming management. In *Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail*, pages 11/1–11/25, 1978.

[136] M.M. Lehman. Laws of Software Evolution Revisited. In C. Montangero, editor, *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag, 1996.

[137] M.M. Lehman and F.N. Parr. Program evolution and its impact on software engineering. In R.T. Yeh and C.V. Ramamoorthy, editors, *Proceedings of the 2nd international conference on Software engineering (ICSE'76)*, pages 350–357. IEEE Computer Society Press, 1976.

[138] M.M. Lehman and J.F. Ramil. Software evolution: background, theory, practice. *Information Processing Letters*, 88(1-2):33–44, 2003.

[139] M. Lesk and E. Schmidt. Lex – A Lexical Analyzer Generator. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1975.

[140] Liant Software Corporation. *RM/Cobol Reference Manual, First Edition*, 2005.

[141] B.P. Lientz and E.B. Swanson. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6), 1978.

[142] B.P. Lientz and E.B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[143] P.K. Linos, M. Harman, and B. Korel, editors. *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press, 2004.

[144] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, 2001.

[145] P. Maurer. Generating test data with enhanched context-free grammars. *IEEE Software*, pages 78–88, November/December 2001.

[146] A. von Mayrhauser and A.M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[147] J. McKee. Maintenance as a function of design. In D.J. Frailey, editor, *1984 National Computer Conference*, volume 53 of *AFIPS Conference Proceedings*, pages 187–193. AFIPS Press, 1984.

[148] L.E. McMahon. SED – A Non-interactive Text Editor. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1978.

[149] Merriam-Webster Online Dictionary. http://www.m-w.com.

[150] Micro Focus. http://www.microfocus.com.

[151] Micro Focus. ObjectCobol. http://www.microfocus.com.

[152] Micro Focus International Limited. *Micro Focus Cobol language reference*, 2002.

[153] Micro Focus International Limited. *Micro Focus Server Express Program Development*, 2002.

[154] J.C. Miller. Structured Retrofit. In G. Parikh, editor, *Techniques of Program and System Maintenance*, pages 179–180. QED Information Sciences, 1988.

[155] L. Moonen. Generating Robust Parsers using Island Grammars. In R. Koschke, E. Burd, and P. Aiken, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE Computer Society Press, 2001.

[156] P. Oman. Maintenance Tools. *IEEE Software*, 7(3):59–65, 1990.

[157] T.J. Ostrand and E.J. Weyuker. The distribution of faults in a large industrial software system. In A. Bertolino and P. Frankl, editors, *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA'02)*, pages 55–64. ACM Press, 2002.

[158] G. Parikh. Exploring the world of software maintenance: what is software maintenance? *SIGSOFT Software Engineering Notes*, 11(2):49–52, 1986.

[159] H. Partsch. *Specification and Transformation of Programs*. Text and Monographs in Computer Science. Springer-Verlag, 1990.

[160] J. Pearsall, editor. *The Concise Oxford English Dictionary*. Oxford University Press, tenth edition, 2002.

[161] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:295–341, 1987.

[162] Pervasive Software. http://www.pervasive.com.

[163] T. Pigoski. *Practical Software Maintenance*. John Wiley and Sons, 1996.

[164] RainCode. http://www.raincode.com.

[165] Reasoning. Software Refinery. http://www.reasoning.com.

[166] T. Reps and T. Teitelbaum. The synthesizer generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE)*, pages 42–48. ACM Press, 1984.

[167] T. Reps and T. Teitelbaum. *The synthesizer generator: a system for constructing language-based editors*. Springer-Verlag New York, Inc., 1989.

[168] R. Rubin. "GOTO considered harmful" considered harmful. *Communications of the ACM*, 30:195–196, 1986.

[169] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[170] R.L. Schwartz and L. Wall. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[171] A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, 45(2–3):193–243, 2002.

[172] A. Sellink and C. Verhoef. Native Patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 89–103. IEEE Computer Society Press, 1998.

[173] A. Sellink and C. Verhoef. An Architecture for Automated Software Maintenance. In D. Smith and S. Woods, editors, *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, pages 38–48. IEEE Computer Society Press, 1999.

[174] A. Sellink and C. Verhoef. Scaffolding for software renovation. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 161–172. IEEE Computer Society Press, 2000.

[175] Semantic Designs. http://www.semanticdesigns.com.

[176] Siber Systems. http://www.siber.com.

[177] Siemens Nixdorf Informationssyteme AG. *Cobol 85 Cobol Compiler Version 2.2A Reference Manual*, 1996.

[178] H.M. Sneed. Architecture and Functions of a Commercial Software Reengineering Workbench. In P. Nesi and F. Lehner, editors, *2nd Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 2–10. IEEE Computer Society Press, 1998.

[179] H.M. Sneed. Risks Involved in Reengineering Projects. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 204–211. IEEE Computer Society Press, 1999.

[180] H.M. Sneed. Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment. In R. Holt, A. De Lucia, and K. Kontogiannis, editors, *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 167–175. IEEE Computer Society Press, 2001.

[181] H.M. Sneed. Integrating legacy Software into a Service oriented Architecture. In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors, *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 2–14, 2006.

[182] H.M. Sneed and P. Brössler. Critical Success Factors in Software Maintenance – A Case Study. In S.L. Pfleeger, C. Verhoef, and H. van Vliet, editors, *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 190–198. IEEE Computer Society, 2003.

[183] H.M. Sneed and S.H. Sneed. Creating Web Services from Legacy Host Programs. In S. Tilley and K. Wong, editors, *Proceedings of the 5th International Workshop on Web Site Evolution (WSE'03)*, pages 59–65. IEEE Computer Society Press, 2003.

[184] Software Improvement Group. http://www.sig.nl.

[185] Software Technology Support Center, Restructurer Tools List, 1999. Available at http://www.stsc.hill.af.mil/.

[186] G. Stap. Cobol Data Flow Restructuring. Master's thesis, Vrije Universiteit Amsterdam & University of Amsterdam, 2005.

[187] M.-A. Storey, E. Stroulia, and A. van Deursen, editors. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society Press, 2003.

[188] Stratego/XT. http://www.program-transformation.org/Stratego.

[189] E.B. Swanson. The dimensions of maintenance. In R.T. Yeh and C.V. Ramamoorthy, editors, *Proceedings of the 2nd international conference on Software engineering (ICSE'76)*, pages 492–497. IEEE Computer Society Press, 1976.

[190] R.H. Thayer. Software Maintenance. *IEEE Software*, 22(4):103, 2005.

[191] M. Tomita. *Efficient Parsing for Natural languages – A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[192] TXL. http://www.txl.ca.

[193] N. Veerman. Revitalizing modifiability of legacy assets. In M. van den Brand, G. Canfora, and T. Gyimóthy, editors, *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 19–29. IEEE Computer Society Press, 2003.

[194] N. Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.

[195] N. Veerman. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*, 57(2):129–163, 2005.

[196] N. Veerman. Automated mass maintenance of a software portfolio. *Science of Computer Programming*, 62(3):287–317, 2006.

[197] N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice & Experience*, 36(14):1605–1642, 2006.

[198] C. Verhoef. The Realities of Large Software Portfolios.
Available at www.cs.vu.nl/˜x/lsp/lsp.pdf.

[199] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9:315–336, 2000.

[200] C. Verhoef. Quantitative IT portfolio management. *Science of Computer Programming*, 45(1):1–96, 2002.

[201] C. Verhoef. Quantifying the Value of IT-investments. *Science of Computer Programming*, 56(3):315–342, 2005.

[202] E. Verhoeven. COBOL island grammars in SDF. Master's thesis, Informatics Institute, University of Amsterdam, 2000.

[203] J.J. Vinju. Personal communication, October 2005.

[204] J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, Universiteit van Amsterdam, 2005.

[205] G. Visaggio, A.K. Amschler Andrews, and F. Lanubile, editors. *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*. IEEE Computer Society Press, 2004.

[206] E. Visser. Scannerless Generalized LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[207] E. Visser. A Survey of Strategies in Program Transformation Systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, pages 1–35. Elsevier Science Publishers, 2001.

[208] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, 2004.

[209] M. Vittek. Refactoring Browser with Preprocessor. In M. van den Brand, G. Canfora, and T. Gyimóthy, editors, *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 101–110, 2003.

[210] M.P. Ward. Assembler to C Migration Using the FermaT Transformation System. In K. Bennett, H. Yang, and L. White, editors, *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 67–76, 1999.

[211] M.P. Ward. Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations. *Science of Computer Programming, Special Issue on Program Transformation*, 52(1–3):213–255, 2004.

[212] G. Weinberg. *Quality Software Management: Volume 1 Systems Thinking*. Dorset House, 1992.

[213] J. Wessler et al. *COBOL Unleashed*. Macmillan Computer Publishing, 1998.

[214] V.L. Winter. *Proving the Correctness of Program Transformations*. PhD thesis, University of New Mexico, USA, 1994.

[215] H. Zaadnoordijk. Source code transformations using the new ASF+SDF Meta-Environment. Master's thesis, Universiteit van Amsterdam, 2001.